RADC-TR-88-206
Final Technical Report
October 1988

A KNOWLEDGE DICTIONARY SYSTEM FOR SCHEDULING SUPPORT

Linguistic Research Institute, Inc.

Peter G. Ossorio and Lowell S. Schneider

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-205 has been reviewed and is approved for publication.

APPROVED:

PATRICIA M. LANGENDORF
Project Engineer

APPROVED:

WALTER J. SENUS
Technical Director
Directorate of Intelligence & Reconnaissance

FOR THE COMMANDER:

JAMES W. HYDE III
Directorate of Plans & Programs

...y Codes
...and/or
...Special
A-1

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (IRD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

A

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
N/A

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-88-206

6a. NAME OF PERFORMING ORGANIZATION
Linguistic Research Institute, Inc.

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
Rome Air Development Center (IRDW)

6c. ADDRESS (City, State, and ZIP Code)
5600 Arapahoe Ave, Suite 206
Boulder CO 80303

7b. ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Rome Air Development Center

8b. OFFICE SYMBOL (If applicable)
IRDW

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F30602-87-C-0067

8c. ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
65502F	3005	RA	80

11. TITLE (Include Security Classification)
A KNOWLEDGE DICTIONARY SYSTEM FOR SCHEDULING SUPPORT

12. PERSONAL AUTHOR(S)
Peter G. Ossorio and Lowell S. Schneider

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM Jun 87 TO Feb 88

14. DATE OF REPORT (Year, Month, Day)
October 1988

15. PAGE COUNT
290

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	04	
15	01	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Expert Systems, Artificial Intelligence, ~~AI~~, Database Scheduling, Project Management, Knowledge Base

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

All social systems must deal with the problem of integrating what's actually happening with what they believe should be happening. A standard system "for expressing general common-sense knowledge for inclusion in a general database... [McCarthy]" is needed for the effective application of expert systems to this task. This project investigated whether a "Knowledge Dictionary System" (KDS) based on State of Affairs (SA) [Ossorio] can achieve this result. The investigation was performed with respect to the reference problem of managing the knowledge necessary to perform and analyze complex scheduling. The conclusion reached was that a KDS knowledge base could be implemented as a set of database relations that capture the part-whole characteristics of schedules; and that a small set of second order relational operators, principally closure, could be combined to achieve a complete part-whole inference logic for supporting fragmentary scheduling at any level and complex dependencies within and across levels including "what if" analyses.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
PATRICIA M. LANGENDORF

22b. TELEPHONE (Include Area Code)
(315) 330-3126

22c. OFFICE SYMBOL
RADC (IRDW)

DD Form 1473, JUN 86 — Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PROJECT SUMMARY

All social systems, from the Government to small business enterprises must deal with the problem of integrating what's actually happening with what they believe should be happening. A standard system "for expressing general commonsense knowledge for inclusion in a general database...[McCarthy]" is needed for the effective application of expert systems to this task. A Phase I SBIR indicates that a "Knowledge Dictionary System" (KDS) based on State of Affairs (SA) [Ossorio] can achieve this result. The objective of Phase II is a fully functional prototype KDS applied to the problem of managing the knowledge necessary to perform and analyze complex scheduling. The KDS consists of SA descriptions expressed as a relational database schema and operators expressed as extensions of relational algebra. The schema is a collection of Basic Process Units (BPU) and Basic Achievement Units (BAU) which capture the part-whole characteristics of schedules. The operators achieve a complete part-whole inference logic. The KDS will support (a) fragmentary scheduling at various levels simultaneously but within a single conceptual framework of contingencies; (b) representing these contingencies within, between and across levels; and (c) propagating the effects of favorable and unfavorable deviations throughout the schedule, including hypothetical "what if" deviations. A KDS for formulating, analyzing, monitoring and revising schedules is immediately applicable to Government and commercial organizations that engage in Government contracting where complex part-whole relationships and cross-project dependencies are involved.

TABLE OF CONTENTS

Appendices

LIST OF EXHIBITS

1. Executive Summary.

During a Phase One Small Business Innovation (SBIR) Research Contract we investigated the feasibility of developing a Knowledge Dictionary System based on State of Affairs technology. We initially chose the general problem of developing and monitoring the schedule of a large project to facilitate the investigation. We then added technical detail regarding a specific schedule, the problem of interfacing computer operating systems (OS) with local area networks (LAN).

The investigation established that State of Affairs technology is a feasible foundation for a Knowledge Dictionary System capable of managing knowledge and the relationships among knowledge of different types and at different levels of detail. The knowledge repository is a conventional relational database management system. The Knowledge Dictionary System is able to interact with the data dictionary of the database to make conventional data available to knowledge-based inference. The goal of the SBIR, to demonstrate the feasibility of straightforward interaction between expert system and database was achieved.

A Phase II effort to build a Knowledge Dictionary System fully capable of managing complex scheduling is feasible.

This Phase II effort can reasonably be expected to lead to a successful commercial product, and be readily extensible to other situations where inferencing is to be performed using data in conventional relational databases.

The scheduling problem undertaken is best exemplified by a Government project in which: (1) some of the requirements are met by Government actions; (2) other requirements are met by contracts; (3) some requirements are met partially by Government actions and partially by two or more contracts with different contractors; and (4) requirements themselves change during the project. Moreover, internal to contracts and Government actions, there are options which sometimes have impact on other options and sometimes impact externally on other contracts or contract options. The Knowledge Dictionary System will provide a standard and comprehensible means to represent and track these complex relationships.

The scheduling problems that will be facilitated by a Knowledge Dictionary System include: (1) building an integrated and detailed overall project schedule from initial statements of contract relationships, Government provided capabilities and individual contract schedules; (2) identifying real or potential conflicts and flagging them for attention; (3) tracking requirements across contracts and identifying what must change when requirements change; (4) determining the impact of schedule changes in one contract on other contracts and Government actions; (5) dynamically assessing the difference between the current schedule and what is actually happening; (6) finding and evaluating options when the difference between schedule and actuality becomes unacceptable; and (7) rebuilding the overall project schedule when options are implemented and programs are modified.

In order to effectively solve these problems, it was concluded that a Knowledge Dictionary System applied to the problem of managing the knowledge necessary to perform and analyze complex scheduling must represent the part-whole characteristics of schedules and provide support for a richer set of inferences that encompass these characteristics. It must allow fragmentary scheduling at various levels simultaneously within a single conceptual framework. It must allow the representation of and support inferencing upon contingencies between and across levels as well as within a single level. It must accept reports of activities and accomplishments at any level and propogate the effects of both favorable and unfavorable deviations throughout the overall schedule. And it must automatically provide the capability to do "what-if" analysis.

Consequently, the scheduling problem has the complexity necessary to serve as an appropriate demonstration problem for Knowledge Dictionary System development. At the same time, the solution to the scheduling problem will provide the Air Force with significant and needed capability in its own right.

2. Definitions.

This report makes use of terminology which has unusual or more specific meaning in the context of State of Affairs descriptions in general, and those of scheduling in particular. This section provides definitions for these terms.

2.1. Social Practice. A social practice is a pattern of behaviors engaged in by persons to cooperatively achieve a desireable state of affairs or to prevent a state of affairs from deteriorating to one of a less desireable sort. Furthermore, the social practice does not succeed by luck or accident, but is taught, learned, and done by persons; and is done successfully by persons routinely enough to warrant a description. The descripton does not specify which actual behaviors to engage in on any given occasion, but codifies what must be accomplished at each stage of the practice, the set of optional behaviors that can lead to that accomplishment, and the rules for determining who is elibigle to engage in the various roles of each optional behavior if it chosen.

2.2. Scenario. When a social practice actually does occur, a description of its occurence (or possible occurrence) is called a scenario. A scenario description differs from a social practice description in that, instead of optional behaviors and eligibility rules, the scenario description specifies which of the optional behaviors are engaged in and who, in particular, engages in them. An actual (as opposed to merely possible) scenario description is generally not available until after the fact and, as such, the report makes reference to "historical particulars" as the behaviors and elements of a social practice that actually occurs.

2.3. Community. A community is a any group of persons who share a common world (i.e., agree upon a set of constituent objects, processes, events and states of affairs such as the "world of golf") and engage in, among other things, a set of fundamental social practices that maintain the community. These include, at a minimum, the practices of conveying the status of *member* (or *non-member*) of the community, negotiating status within the commmunity, and using the language (jargon) of the community.

2.4. Institution. An institution is an organized set of social practices that is of sufficient import to the community that memebers decide to formally record its description as a set of rules and eligibilities and to include the maintenance of the description as a part of the social practice itself; i.e., it is given a "life" of its own to insure that it will survive any or all of its members.

2.5. Scheduling (1). All use of the term "scheduling" or "schedule" outside of Sections 4 and 5 is with the connotation that obviously comes to mind and needs no further explication for the purpose of this report.

2.6. Scheduling (2). Secondly, "scheduling" is used within Section 4 to denote the entire social practice of planning some achievement, monitoring the progress toward that achievement including the monitoring and revision of inconsistencies among **all** elements in the practice, and actually attaining some achievement and comparing it with that which was planned. Formally speaking, scheduling used in this way denotes the **entire process** of transition from the social practice description to the scenario description.

2.7. Scheduling (3). Finally, within Section 5, the formal social practice and scenario descriptions actually used in the investigation was the institution of scheduling as it occurs in DoD when large and complex systems a. e developed and includes the formal stages of PDR, CDR, TRR, etc., and their attendant reporting requirements. The difference between scheduling (3) and scheduling (2) is that it includes formalities established by the DoD scheduling community (most importantly, the language of scheduling in that community) whereas the latter focuses on the "essence" of scheduling that could be institutionalized by a community but hasn't been.

2.8. Needs. In the context of any social practice, there are basic needs which, if not met, make that practice impossible to engage in. In the context of scheduling we thus define the concept of need as something, which if not met, will result in a pathalogical case of the schedule. The following are the minimal needs for a schedule to be carried out.

2.8.1. Status. This refers to the place within a schedule that a person or thing has and, operationally, determines what behaviors it is eligible to be a part of. Without status, nobody or nothing is eligible to be any element of a schedule and thus the schedule cannot be carried out in practice. This also implies that the social practice of making status assignments is integral to the social practice of scheduling.

2.8.2. Distinction. To make a distinction is to in roduce order and meaning in a schedule by, at least, classifying things as being of one sort rather than another. If it is not possible to distinguish between the achievement of a given schedule and some other achievement, it is impossible to determine if the schedule exists let alone to engage in it.

2.8.3. **Adequacy.** All stages of a schedule must have at least one optional behavior that an eligible person is competent to perform, and for which all the elements have at least one eligible individual. If there is no such adequate behavior then the schedule will, at best, miscarry.

2.8.4. **Value.** There must exist some principles by which one achievement can be deemed to be more or less desireable than another. At a minimum, there needs to be hedonic, prudential and ethical principles and, in general, there will also be esthetic principles. Without values, there are no means by which an achievement can be evaluated and, hence, no motivation for engaging in any schedule that achieves it.

2.9. **Conflict.** In this report, conflict denotes any state of affairs in which one or more needs are not being satisfied. When it is said, as it will be throughout this report, that a schedule is in conflict, it means, first of all, that one or more of any of the needs is not being met and it's use in this sense is much broader than the simplistic calendar conflict that arises in the commonplace usage of the term scheduling, although that is a genuine example of conflict as we use the term. Secondly, it means that the schedule is in conflict in one or more of the ways in which it could be in conflict according to its description. Its use in this sense is potentially narrower than in commonplace scheduling in that if a particular way of being in conflict is not directly or indirectly present in the description, conflicts of that kind will never arise (e.g., unless there is an explicit constraint that a person can only be in one place at a time, a schedule that calls for a person to be at two different places at the same time will not be in conflict).

2.10. Knowledge. In this report, knowledge refers to that which is necessary to engage in scheduling. More specifically, in this usage knowledge includes, at a minimum, for the social practice of scheduling: (a) Want; i.e., the knowledge of the state of affairs that is desired as a result of engaging in the schedule; (b) Competence; i.e., the knowledge of which behaviors are called for by the schedule and how to engage in those behaviors; and (c) Cognition; i.e., the knowledge of the present state of affairs (including knowledge of limited competence and want) and what social practices are available to engage in to alter (maintain) the present state of affairs.

2.11. **Knowledge Dictionary.** A medium or system within the context of a given social practice for recording the competence and cognition knowledge necessary to engage in that social practice in such a way that a person that wants to engage in that social practice but lacks some of the knowledge necessary to do so may use the knowledge dictionary to supply the knowledge that he lacks. In addition, a knowledge dictionary may also include the want component of the knowledge but that a persons ability to make use of the want component is severly limited in that if he lacks that knowledge, he cannot, in principle, know that he wants the knowledge in the dictionary in the first place.

2.12. **Knowledge Base.** As opposed to a knowledge dictionary which is merely the medium or system for recording knowledge, the knowledge base refers to the knowledge that is actually recorded in the knowledge dictionary at the moment; i.e., the range of social practices identified and the depth to which they are defined.

2.13. **Part-Whole Relation.** Social practices in general, and schedules in particular, are primarily described by part-whole and part-part (see below) relations. The description of a whole schedule is always an elliptical (incomplete) description for which a more detailed description could be provided by decomposing it into sub-tasks.

Conversely, schedules can be combined to yield a more comprehensive whole schedule. In either case, the result is to create a relationship between a part and the whole of which it is a part, i.e., a part-whole relation, for which both components are necessary to the existence of each **as parts and wholes** (they may all exist independently and usually do). If the whole does not exist, then nothing is eligible to be a part of it. If any of the parts does not exist, it is impossible to construct the whole from its parts.

2.14. Part-Part Relation. When a part-whole relation is created, the potential exists to have part-part relations. A part-part relation is a contingencey statement that declares which combinations of parts may occur concurrently as parts of a whole.

2.15. Inference. In this report, inference is used to denote the process by which one or more scenarios can be formulated from a social practice description. More specifically, as used in scheduling, inference denotes the outcome or range of outcomes that can be reasonably inferred from the schedule given its present state. At the outset, the schedule is merely a social practice that has yet to be engaged in and thus no possible outcomes exist. As the schedule progresses; i.e., as options and elegibilities are replaced with historical particulars, the range of possible outcomes is constrained. A complete description of the actual outcome can never inferred until after the fact, but informative incomplete descriptions can be inferred at any time once the schedule is engaged in.

2.16. Inference Engine. In this report, the inference engine(s) denote an algorithm and its implementation as a computer program which is capable of inferring particular aspects of possible outcomes of a social practice, a schedule in particular, given only limited cognitive knowledge of the present state of affairs. In some cases the term is used in the singular without qualification to denote the aggregate inferencing mechanism that infers every aspect of the possible outcomes, hence the outcome itself. In other cases, it is used in the qualified plural to denote the mechanism that infers some aspect of the possible outcomes based on a particular kind of knowledge.

2.17. Part-Whole Inference. In this report, part-whole inference is used to denote the kind of inference that can be formulated from part-whole and part-part relations. Part-Whole inference is fundamentally different than logical (first-order predicate logic) inference in one important way. Logical inference is serial in nature and always complete. I.e., given

Consequent if $Antecedent_1$ and ... and $Antecedent_n$

the consequent is true only indirectly by virtue of all the antecedents being true. And if any of the antecedents are, themselves, consequents then it is true only indirectly by virtue of its antecedents being true, ad infinitum. It is only where antecedents have no further antecedents, called ground clauses, that direct assignment of truth values can be made. By contrast, part-whole inference is parallel in nature and always incomplete. Truth status can be directly assigned at any level of the part-whole decomposition; i.e., at the level at which it was actually observed, and the truth status of a whole at any level determines the truth status of its parts (and their parts, ad infinitum) and is always maintained as the set of parts whose existence needs to be confirmed.

2.18. Small Business Innovation Research (SBIR). The SBIR program exists in all Government agencies to encourage small businesses that do not have the capital to maintain their own research and development programs to create innovative and commercially viable products. Throughout this report there are references to the Phases (I, II, III) of an SBIR contract. Normally, a Phase I SBIR effort is a small (6 man-month) contract with the Government to establish that an innovative concept is sound, feasible, and commercially viable as the basis for a product. A Phase II

SBIR effort is a larger (2 man-year) contract with the Government to actually develop a prototype of the product for hands-on testing and evaluation. A Phase III SBIR effort is a full-scale project, funded from private sources attracted by the results of Phase II, to develop and market the product in the commercial sector.

3. Introduction.

All social systems, from the Government to corporations to small business enterprises must deal, on a daily basis, with the problems of integrating what's actually happening with what they believe should be happening. A standard and comprehensible system "for expressing general commonsense knowledge for inclusion in a general database...[1]" could, to a large extent, relieve the present need for training in a vast number of social practices while concurrently enhancing the expertise of individuals participating in those practices. During a Phase I SBIR Contract [2] we investigated the feasibility of developing a Knowledge Dictionary System based on State of Affairs [3] technology to achieve this result. To facilitate the investigation, we chose a specific social practice that appears to be paradigmatic in all enterprises: the practice of developing and monitoring the schedule of a large project. Typically, such projects are complex, extend over lengthy periods of time, involve the meshing of a variety of components and activities at different levels, and, most importantly, do not proceed according to plan. It is these characteristics that make managing the knowledge necessary to perform and analyze complex scheduling a difficult problem within the confines of existing technology [4].

The problem is best exemplified by a Government project in which: (a) only some of the requirements are met by Government actions; (b) other requirements are met by contracts; (c) often, a requirement is met partially by Government actions and partially by two or more contracts with different contractors; and (d) the requirements themselves change at discrete points of time during the project. Moreover, internal to contracts and Government actions, there are options which have impact on other options, and sometimes impact externally on options in other contracts. The consequence of not having a standard and comprehensible means to represent these complex relationships presents numerous problems in at least the following tasks:

(a) receiving initial statements of contract relationships, government provided capabilities, and individual contract schedules and integrating these into an overall project schedule;

(b) identifying real or potential conflicts and flagging them for attention;

(c) tracking requirements across contracts and identifying what must change when requirements change;

(d) determining the impact of schedule changes in one contract on other contracts and Government actions;

(e) dynamically assessing the variance between the current schedule and what is actually happening; and

(f) finding and evaluating options when the variance becomes unacceptable and rebuilding the overall project schedule when options are implemented.

In summary, managing the knowledge necessary to perform and analyze the scheduling of large, complex projects is a significant problem throughout Government and private enterprise. Yet the tools available to deal with this problem, no matter how elaborate, remain limited in their ability by virtue of having the same technological foundation, one that has not changed for many decades.

An effective Knowledge Dictionary System applied to the problem of managing the knowledge necessary to perform and analyze complex scheduling must, at the foundation level, represent the part-whole characteristics of schedules and provide support for a richer set of inferences that encompass these characteristics. It would allow fragmentary scheduling at various levels simultaneously but within a single conceptual framework of contingencies. It would allow the representation of these contingencies, not only within a single level, but also between and across levels. And it would accept reports of activities or accomplishments at any level and propagate the effects of both favorable and unfavorable deviations throughout the overall schedule, including hypothetical "what if..." deviations.

The Phase I investigation revealed that State of Affairs (SA) is a feasible means to represent the knowledge necessary to schedule analysis. The essence of this representation is to view a schedule as the part-whole decomposition of a process, most of which, at the time of analysis, has not yet unfolded and is not yet available as observable information. Specifically, the representation of a schedule must capture at least the following characteristics of schedules.

(a) A schedule, being a description of a process, divides into smaller, related schedules without limit.

(b) An objective can be achieved by any number of optional processes, each of which has a schedule.

(c) The options available to achieve an objective within a schedule are contingent on the options chosen to achieve other objectives in that schedule.

(d) If an option in a schedule requires a resource, then its viability is contingent on other options in that schedule requiring that resource.

(e) If an option has temporal interrelationships with other options in a schedule, then its state (beginning, occurring or ending) is contingent on the state of those options.

(f) When options in a schedule are in conflict due to resource requirements or temporal interrelationships, the process by which the conflict occurred is an (undesirable) option at some level of that schedule.

(g) The achievement of an objective in a schedule is equivalent to the occurrence of some option for every process necessary to that achievement.

(h) What may be reported as a process needs to be re-describable as the achievements resulting from the options that constitute that process; and what may be reported as an achievement needs to re-describable as the options for the processes that lead to that achievement.

The investigation also revealed that the kinds of inference that can be performed over part-whole representations of schedules are considerably richer and more complex than those presently available and that all of these, once the part-whole representation is in place, are feasible to implement as computer-based inference algorithms. Specifically, the inference algorithms applied to part-whole representation of schedules must satisfy the following objectives.

(a) The part-whole inference process must be augmented to recognize that some achievements in a schedule may be safely assumed to exist for long periods of time while others must monitored at very short intervals. [5]

(b) Frequently, it is the case that one of two (or more) different achievements in a schedule may have occurred given the reported information about the progress toward them, but due to real world resource limitations, it could not be the case that more than one is possible concurrently [6]. When this occurs, the process of part-whole inference must be able to:

(1) recognize that a partial resource conflict

exists and has the potential to create a pathological state of the schedule;

(2) continue to analyze both possible states of the schedule in terms of other part-whole resource relationships; and

(3) when a total resource conflict exists, support the application of choice principles to determine which state of the schedule is more desireable.

(c) It is also frequently the case that only one of two or more different achievements in a schedule can occur at a given time due to temporal conflicts. [7] Moreover, these may not be simply sequential relationships but may also involve complex overlapping temporal relationships. In such situations, the process of part-whole inference must be able to incorporate temporal chains of reasoning that:

(1) recognize that a partial temporal conflict exists and has the potential to create a pathological state of the schedule;

(2) continue to analyze both possible states

of the schedule based on other part-whole temporal relationships; and

(3) when a total temporal conflict exists, support the application of choice principles to determine which state of the schedule is more desireable.

(d) In the case of both resource and temporal conflicts, the support of choice principles must be provided in at least two ways:

(1) choice principles may be represented in advance as descriptions of several versions of a schedule (e.g., aggressive, conservative) in which case the most desireable version will emerge automatically (i.e., no conflicts will exist for one of the versions);

(2) choice principles may be applied ad hoc in which case the part-whole inference process must be able to provide notification of where in the schedule these conflicts exist.

In either case the part-whole inference process must support removing a version of the schedule and determining the consequences of eliminating its associated conflicts.

(e) Perhaps the most critical requirement is that the process o part-whole inference must be able to draw inferences from an admixture of report types affecting different levels in the schedule.

(1) In the first case the process must be able to infer that multiple reports of different types are, in fact, reports about the same (or part of the same) progress in the schedule and then use these equivalences in its chain of reasoning. [8]

(2) In the second case, it must be necessary to establish the existence of parts from wholes as well as conversely. Most familiar is the induction that if all the prerequisites to an achievement are complete, then the milestone has been reached. Less familiar, and unique to part-whole inference, is the deduction that the completion of an achievement implies that every prerequisite is complete (and some of those may be prerequisite to other achievements). [9]

Moreover, allowing heterogeneous reporting not only integrates the formal reporting systems already in place, but also taps less structured sources of information (meetings, conversations, etc.) that already exist but are rarely incorporated into the scheduling process in any formal way.

In summary, the Phase I investigation revealed that a system which effectively addresses the problem of managing the knowledge necessary to perform and analyze complex scheduling must capture the part-whole characteristics of schedules and provides support for a much richer set of inferences that encompass these characteristics. The system has been shown to be both possible and practical to develop and would have enormous potential throughout both Government and private industry.

4. KDS for Scheduling Support.

This section is divided into three parts. The first presents the concepts and facilities of an innovative Knowledge Dictionary System for scheduling support. The second presents the design requirements that must be satisfied and and the feasibility of their implementation. The third provides a functional overview of a pre-prototype Knowledge Dictionary System based on this concept that was applied to the scheduling problem as an integral part of the Phase I investigation. The knowledge base that was actually developed during the investigation and implemented within the system is presented in the following section.

4.1. Concepts and Facilities.

A Knowledge Dictionary System is comprised of two principal components: the knowledge representation structure; and a part-whole inference engine that operates on this structure. The knowledge structure is a collection of Basic Schedule Units which encapsulate the descriptions of (a) the processes and achievements that comprise the schedule, (b) transformations for redescribing processes as achievements and conversely; and (c) the part-whole relationships among these constituents. The part-whole inference engine navigates among the schedule units to (a) determine the degree to which higher-level schedule units are complete based upon the state of their constituents, (b) propagate the impact of reports about lower-level schedule units to the higher-level units of which they are a part, (c) detect actual or potential conflicts due to either resource or temporal constraints.

4.1.1. Knowledge Structure. Corresponding to the basic concepts of scheduling (process and achievement) is a format for representing concepts of each type. These are the Basic Process Unit (BPU) and the Basic Achievement Unit (BAU). The reason for different units is that the first part of each represents the observational aspect of the concept (the way it was reported: i.e., a process has stages, an achievement has deliverables, etc.) The second part of each format has, embedded within it, the elemental resource aspect of the concept which represents the means of converting from one to the other.

Within the Knowledge Dictionary System the representation units are be recorded as a set of interrelated tables. The reasons for this are twofold. First, from experience with our existing implementation, one will not want to deal with an entire representation unit at once, but rather will want to focus on one or two specific aspects of it (e.g., the list of options for a given stage of a process) at a time. Secondly, readily available Database Management Systems handle tables (relations) with great facility and it is considered to be of great importance that the Knowledge Dictionary System be able to make use of these readily available facilities rather than to depend on its own, unique way of representing knowledge. [10] For illustrative purposes the representation units are presented in this section in their expanded hierarchic form in Tables 1 - 2.

In addition, the fact that separate reports can be formulated as reports of the same (or part of the same) schedule requires that there be logical relationships among the different representation units. These are represented within the Knowledge Dictionary System as a set of Transition Rules. These are systematized as follows:

T1) A schedule is a totality of related processes and/or achievements and/or schedules.

T2) A schedule (or process or achievement) is a constituent of a schedule.

T3) An achievement is a schedule that has other, related achievements as immediate constituents.

T4) A process is a sequential change from one achievement to another.

T5) A process is a schedule having other, related processes as immediate constituents.

T6) An achievement is a direct change from one schedule to another.

T7) An achievement is a schedule having two schedules as constituents (i.e., "before" and "after").

T8) A given schedule's having a given relationship to a second schedule is a schedule.

T9) That a process begins is an achievement and that it ends is a different achievement.

T10) That a process occurs (begins and ends) is a schedule having three schedules as constituents (i.e., "before," "during," and "after").

In addition to these, because of their inherent and necessary recursion, it is also necessary to have limiting cases that can be invoked to "stub off" the unlimited decomposition or composition permitted by the rules. These are:

L1) The schedule which includes all other schedules (i.e., the master plan).

L2) An achievement that has no constituents, hence is an atomic particle (i.e., an indivisible resource).

L3) A process that has no constituents, hence no beginning that is distinct from its end (i.e., the equivalent of an achievement).

TABLE 1 - THE BASIC PROCESS UNIT

P-NameA: The process "Name" of process A

P-DescriptionA: The "Description" of A. It specifies:

I. P-Paradigms: The major varieties of P-NameA. This is a technical convenience. Every process has at least one paradigm. But many processes can occur in ways so different that it is easier provide a separate description for each rather than to encumber a single description with an unmanageable number of contingencies. For each paradigm, the following are specified:

(a) Stages 1-K: These are "Names" of sequentially distinct progressions within A. They are systematically specified as P-NameA11, P-NameA12, ... ,PName-A1K for Paradigm 1. For each stage are specified:

(1) Stage-Sets 1-M: These are the "Names" of the subsets of a stage that have temporal interrelationships that are not merely sequential and are distinct in that each element of the set corresponds to a definable achievement. They are systematically specified as P-NameA111, P-NameA112 for stage 1. For each stage-set the temporal relationships among the elements are defined.

(2) Options 1-N: These are the "Names" of various exemplars of the stage in question, i.e., the various ways in which that the progression could occur (every stage has at least one option). Each Option is systematically specified as P-NameA111, PnameA112, ... ,PnameA11N. For each option is specified a set of processes, which, if each were completed, would result in an achievement that would qualify as the progression denoted by the stage. These are processes in their own right and thus any further specification is accomplished by another BPU.

(b) Elements: These are the logical categories within the process for the resources necessary to the occurrence of the process.

(c) Individuals: These are the formal exemplars of Elements that are actually present or available at the time the process occurs.

(d) Eligibilities: These are the constraints that exemplify which individuals are capable of being which elements in an occurrence of the process.

(e) Contingencies: These express co-occurrence constraints among the stage-set-options of the process (i.e., the options available at one stage of the process are contingent on which options may have been selected at another stage - and it is these, when they become too complex, that motivates the employment of different paradigms).

(f) Versions: This is the net effect of all of the above. It captures the result that the different versions of the process P-NameA on different occasions need not resemble one another in any way other than their being alternative versions of P-NameA.

TABLE 2 - THE BASIC ACHIEVEMENT UNIT

A-NameA: The achievement "Name" of achievement A

A-DescriptionA: The "Description" of A-NameA. It specifies:

I. A-Paradigms: The alternative decompositions of A-NameA. This is a technical option. If only one paradigm exists, it will be the same as A-NameA. For each paradigm, the following are specified:

(a) Constituents: A list of immediate constituents which are systematically designated as A-NameA11, A-NameA12, ... ,A-NameA1N for paradigm 1. Each constituent can now be expanded (decomposed) using another Basic Achievement Unit.

(b) Relationships 1, 2, ... ,M: These are given by a list of relationships in which each item on the list is specified as follows:

(1) Name: An expression which identifies an N-place relationship. N may vary among different relationships in the list.

(2) Elements: A list of N elements, each of which is one of the members of the N-place relationship.

(3) Individuals: These are the formal exemplars of Elements that are actually present or available to participate in the relationship when it occurs.

(4) Eligibilities: A specification of which individuals may or must participate as which elements in the relationship by virtue of their constituency in A-NameA1.

(5) Contingencies: These express constraints of two kinds; attributional constraints on the elements of the relationship, and co-occurrence constraints among the elements of the relationship.

4.1.2. **Part-Whole Inference Engine.** The Knowledge Dictionary System, by virtue of its knowledge representation structures, is able to continually analyze the part-whole character of a schedule, most of which has not yet occurred or is not yet available as reportable information. It does this routinely, in much the same way as a person is subconsciously aware of what he thinks is going on, and continually re-assesses that conclusion in light of new developments as he becomes aware of them. And while the system, itself, does not impose any rigid distinctions among the levels of part-whole characterizations (the system treats this as a continuum), it is helpful to explain the behavior of the inference engine by reference to three levels: (a) the "master plan" level which represents an overall pattern of activity directed at the achievement of the project, most of which has not yet occurred but which will serve as an explanation for what did occur after the fact; (b) the reporting level which represents situation--dependent facts (empirical identities) as they occur, without regard for how those facts may or may not fit into a higher level pattern; and (c) a logistical level in which facts reported in one context of the schedule can be redescribed as facts in the another context of the schedule.

4.1.2.1. **Master Plan Level.** The goal of the inference engine at the master plan level is to determine that the plan is or is likely to be completed as described. This is accomplished by a process termed "back--chaining" in which the inference engine determines which constituents of the plan are essential to its being completed. In turn, each of those constituents is similarly analyzed and so on until the inference engine can establish that, in this situation, an empirical individual is available and eligible to be each element of each constituent. If it succeeds, it records the results of the analysis and the time [11] at which the conclusion was reached. Each time it fails, it records the constituent that it failed to materialize, the time that the failure occurred and the reason that the failure occurred, and proceeds to another constituent for which the entire process is repeated. This process is continuous and the result is a record of the chain of reasoning, and the links in that chain where the reasoning failed, so that the inference engine can repeatedly retry to establish each unconfirmed empirical identity until it successfully works backwards (hence, back-chaining) to the master plan from which it started.

4.1.2.2. Reporting Level. The goal of the inference engine at the reporting level is straight-forwardly that of establishing empirical identities and it does this via a process termed "forward-chaining". When the inference engine encounters the report of a fact that is eligible to be an essential element in some level of the schedule (e.g., is an option for a stage of some process), it processes the contingencies that operate as constraints on its eligibility. This process is also continuous in the sense that, each time a new report enters the system, the inference engine checks which options for an element currently under consideration may have become eligible by virtue of that fact, and proceeds to work forward (hence, forward-chaining) in search of new empirical identities. Whenever an empirical identity is established (or discounted), the result and the time that the result was obtained is recorded.

4.1.2.3. Logistical Level. Finally, the goal of the inference at the logistical level is to determine whether empirical identities established in one context of the schedule can be redescribed as empirical identities in another context of the schedule. Empirical identities are cases where one thing is the same thing as another thing, not as a universal or necessary fact (the way "a rectangle" is always and necessarily the same thing as "a trapezoid with right angles") but only as a historical, empirical fact; i.e., cases in which <u>in this situation</u> what is described as P is the same thing as what is described as Q and R, etc. The engine does this by a process termed "cross-chaining". Whenever an empirical identity is established at the observational level the inference engine examines all similar constituents, without regard for what they are constituents of. I.e., it cuts across the chain of inference (hence, cross-chaining) and merely records the result and the time at which it was obtained. It does nothing further at this point. However, when the inference engine is subsequently back-chaining or forward-chaining in another chain of inference it will encounter and make use of these results.

4.2. Design Requirements.

The investigation of how successfully the technology supports scheduling and the practicality of implementing it within existing software technology proceeded along two lines in parallel. The first of these was to transform the knowledge base into a set of database relations and this is discussed first. The second was to determine the requirements for processing the knowledge base in terms of a generalized set of database operations. This represents the "inference" engine.

4.2.1. Knowledge Base Development. The development of the knowledge base is principally the task of developing a reasonably normalized relational schema of the information represented in both the Basic Process Unit (BPU) and the Basic Achievement Unit (BAU). Due the essential similarity between the BPU and BAU, it was anticipated that a single schema will serve both and this, in fact, turned out to be the case. The principle relations developed include the Part-Whole relation, the Element-Individual relation, the Eligibility relation, the Temporal-Constraint relation, the Assignment relation, and the Connectivity relation. Furthermore, although the BPU and BAU provide for stage-sets and option-sets, the normative case is that (a) a stage is singular, and (b) only a single process is necessary to satisfy the occurrence of that stage. The following discussion assumes this simplification for brevity unless otherwise noted. The convention of underlining the domains which comprise the key of the relation is employed.

4.2.1.1. Part-Whole Relation. The notion that the option of a stage of a paradigm of a process is, itself, a process is the essential closure property that has to be captured in the part-whole relation.

(PROCESS-NAME, PARADIGM-NAME, STAGE-NAME, OPTION-NAME)

Note that an OPTION-NAME will have the same status as a PROCESS-NAME which creates the desired closure. If we decompose this into binary relations as follows, we disclose the first major naming problem we have to contend with.

(PROCESS-NAME, PARADIGM-NAME)

(PARADIGM-NAME, STAGE-NAME)

(STAGE-NAME, OPTION-NAME)

(PROCESS-NAME, OPTION-NAME)

The last of these is the relation with the transitive closure property. The problem disclosed is thus. While PROCESS-NAME and OPTION-NAME are clearly unique across an entire schedule (even if that is L1) it is not clear whether or not PARADIGM-NAME and STAGE-NAME are. If they are then the original relation becomes:

(<u>PROCESS-NAME, PARADIGM-NAME, STAGE-NAME</u>, OPTION-NAME)

In addition, each of the key components above has a truth status and some kind of date-time group attached leading to a very large (i.e., wide) relation to be maintained and processed. This will cause enormous efficiency problems in the chaining process since the inference engine wants a very compact relation so it can load huge restrictions of it into fast memory.

The obvious solution is to have the user maintain (or, if possible, do it for him) a highly encoded identifier for the key and for the OPTION-NAME such as a legal outline notation (e.g., 42.2.1.2). The exception to this is that an OPTION--NAME needs <u>two</u> identifiers: one within the process description; one independent of any process description (since it, of course, can occur in several processes). So we will obtain a relation in which the option is represented twice, once as part of the process and once independently of any superior process; and note that this is a general requirement, in any case, when we deal with option-sets.

(<u>PROCESS-NAME, PARADIGM-NAME, STAGE-NAME, OPTION-NAME,</u>
OPTION-PROCESS-NAME)

Now, since each component of the key needs a truth status (TS) and date-time group (DTG) associated with it; and we want to keep the relation seen by the inference engine as small as possible; that suggests that we need a way to normalize the relation that allows us to <u>not</u> have to maintain all of these concurrently; i.e., once the option is true, we don't need its truth status anymore because the stage is now true, and the same follows as we work up through paradigm and process. Thus, if we construct the code above to represent the key and call it PPSO and call an identical code OPPSO to represent the independent process identifier, we can create a relation as follows:

(PPSO, OPPSO, LEVEL, STATUS, DTG)

where LEVEL refers to what the status applies to; i.e., the process, the paradigm, the stage or the option. Hence, if the level were option and the status was true, the inference engine would know to stop looking at any other options for that stage and start looking for options for the next stage.

We will, of course, want to provide a relation, either separate or with these included, that has the explicative names for benefit of the user and even some explanatory text but the inference engine need not be aware of this. If this is done in a single relation it be slow because the inference engine will have to project out all the text before it can get to work. If separate relations are maintained, it may appear to be a burden to the user but, in fact, is probably not since the existing implementation allows multiple relations, synchronized by a join clause, to appear on the screen concurrently.

4.2.1.2. Element-Individual Relation. While the part-whole relation determines that a process has occurred because at least one option for each of its stages has occurred, the element-individual relation determines that a process is, at least, possible because at least one individual exists for each of its required elements. In some ways this sounds a bit redundant; i.e., there are two ways to reach a conclusion; but, in fact, that is not the case.

In the early drafts of a schedule it may be somewhat confusing to read because the formal elements of the process will be very much interspersed in the process descriptions, while not yet having been recognized as formal elements. Later drafts will read almost entirely like outlines of the process/stage/stage-option (process) decomposition with very few references to elements. And these are easier to read since, if the stage-options are listed (which are, of course, processes in their own right), it is only important that a stage has or hasn't occurred and if all of them have occurred then it's the case that the process has occurred and the elements are no longer relevant.

However, at the bottom of the decomposition there are no stage-options to check. The only criteria available are the elements. I.e., at this level, there are no process descriptions per se but only SA descriptions which place elements and individuals in a one-to-one correspondence via eligibility rules.

The first problem to resolve is, again, the duplicity of names. Every element requires a global (i.e., L1) name as well as a name unique to the (lower-level) process in which it occurs. This is what will facilitate the forward-chaining (bottom-up) that needs to occur when element-individual data changes in the database. Hence, the relation obtained will be very similar to that used to capture the part-whole information:

(<u>PPSO, OPPSO,</u> ELEMENT, INDIVIDUAL)

Note that this provides a list for **every** PPSO of the elements and individuals applicable, even though that PPSO may, in fact, be L1. In addition, it allows that the elements, the eligible individuals may vary from one PPSO to another - an essential requirement. Moreover, since a given element and individual eligible to be that element, are concurrently both independent of and in the context of a particular process at any level, we can capture the L1 description by the relation:

(<u>*,*,</u> ELEMENT, INDIVIDUAL, RULE-NAME)

where the "*" indicates a an L1 (or "null") PPSO or, alternatively, provide explicit values for the PPSO putting the description in the context of a particular process without altering its description in the L1 context [12].

4.2.1.3. **Co-occurrence Relation.** Not surprisingly, a relation that captures constraint rules has a character very similar to that of the part-whole relation. If elements/individual names can be appended with PPSO names to make them specific to a given process, then the same principle can be applied to constraints (this is quite different from the PROLOG approach in which all rules are essentially global). Ignoring generality of rules for the moment, the basic form of the relation is:

(<u>CONSEQUENT-NAME, ANTECEDENT-NAME,</u> STATUS, DTG).

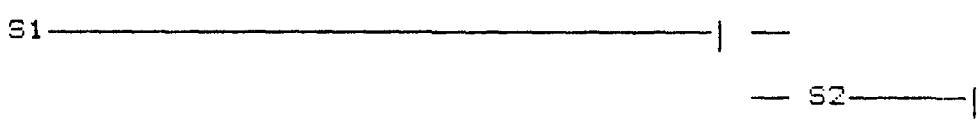
If we embed the PPSO in the names of the ELEMENT and INDIVIDUAL, we again create a situation in which rules can be stated in any context (even L1) independently of any other context - which is, for now, the desired result [13]. To the extent that co-occurrence constraints only involve stage-options [14], the basic form of this relation is, again, that of the eligibility relation with both the CONSEQUENT and ANTECEDENT slots being filled with OPPSO and PPSO names respectively. Hence:

(<u>PPSO, OPPSO,</u> STATUS, DTG).

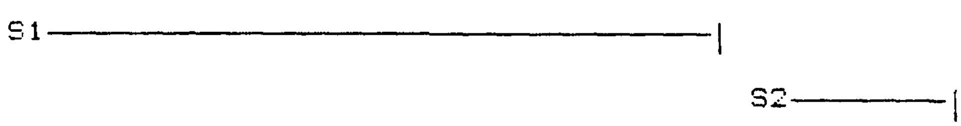
If these are stated **within** the context of a given process, the information is probably derivable from the part-whole relation, although at the expense of performing a projection of that relation. However, this constraint does not allow for an expedient treatment of global (L1) constraints.

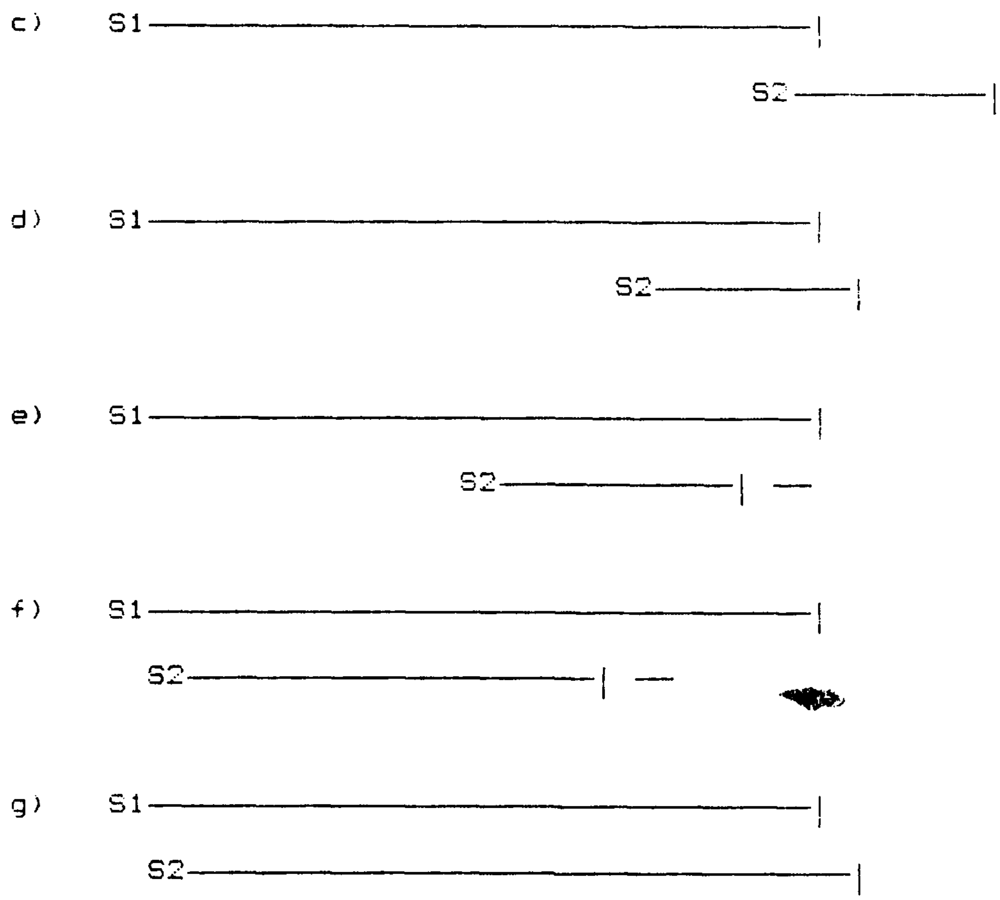
4.2.1.4. Temporal Constraint Relation. As long as we assume that stages occur one after the other, a relation to represent the sequentiality of stages is probably not necessary as we can use some sort of encoding of the PFSO to represent that information. This is probably a very idealistic situation and the principal motivation for stage-sets [15]. *In fact, in the general case, we probably* have an analogy of Gray's concurrency problem [16] which is best illustrated graphically. If S1 represents stage one and S2 represents stage two and they can be of varying durations then we can get at least all of the following cases:

a)



b)





If S2 begins before S1 then we get all of the above diagrams with the names interchanged. Note that the above cases are only for **two** stages. For now, a relation that captures only this would be nice in the hopes that if it works for two it will carry any multi-stage interdependencies but such is not at all obvious yet and needs additional work [17].

The first problem to address is how to represent the information in normal form. At first glance, at least the following information is needed:

a) that the start of S2 is related to the start of S1;

b) the end of S2 is related to the start of S1;

c) that the start of S2 is related to the end of S1;

d) that the end of S2 is related to the end of S1;

e) for each of the above, the kind of relationship, i.e., before, coincident, or after.

This would yield a rather cumbersome relation for each pair of stages that are temporally related:

(PPSO1,PPSO2,2S1S,2E1S,2S1E,2E1E)

in which ySxS encodes whether y Starts before, coincident with, or after x Starts, etc. This allows a stage to be pair-wise temporally dependent on any number of other stages but does not, as previously stated, allow n-way dependencies among stages. Another approach that might work (and would certainly make the relation more tractable) would to be replacing the four endpoint relationships with a single (or possibly two) state relationship; e.g., S2 can only be occurring if S1 is occurring (has begun and hasn't ended).

The two approaches may be equivalent but further analysis is required.

The second problem to tackle is "what to do with the information" once we've got it so we can determine a sensible way to store the information for processing. Whether the processing is a further complication of the part-whole inference engine or a separate process that independently generates data for the part-whole inference engine can be taken up at a later time. For now, we may as well think of it as a separate process since it's certainly performing a drastically different function than part-whole inference.

The essence of the analysis at first glance seems to be that of queuing. The goal is to determine whether or not S2 has occurred (i.e., reached its endpoint) and that is dependent on where S1 is as far as its state is concerned. And, in turn, S1 may be similarly dependent on some other stage(s) [18]. Thus we can visualize each stage of a process, and the state it is in (not begun, beginning, occurring, ending, ended, etc.) as a "resource" for which other stages are competing. I.e., when a stage is analyzed, the list of stage/state combinations it requires to be in a given state is queued up against the stages on which it depends. Then, periodically, the queued stages are compared with the stage/state they are waiting for and those whose requirements are satisfied are dequeued and their state is updated [19].

While there are numerous solutions to queuing than encompass the notion that a resource (or a request for it) can be in numerous states, an approach analogous to Gray's seems a natural starting place (since we began by stating the problem in a way similar to his.

Before proceeding to develop a queue structure and protocol, we need to solve two problems:

a) develop a compatibility matrix of states so that if a request is that a state is occurring and the state is beginning then it is natural to grant the permission to continue (for the waiting stage to move to the next state);

b) victim selection algorithms (these are, essentially, the choice principles) so that if two or more stages are deadlocked then the least likely can be removed from the queue and restarted somewhere else (or if it's already queued somewhere else, then let it proceed there to see what happens).

The first of these seems rather straight forward although we will have to come up with a number of states in which a process can be other than those defined in the transition rules alone. The second seems difficult at best. There are simply too many choice principles (as defined in Place [20]) and their totally content-free nature makes it doubtful whether the system can determine which apply to which situations. Perhaps this is the one problem during the prototype phase that we can "throw over the wall" to the user. I.e., when there is a conflict, the system can point out the candidate victims but the user will make the choice.

4.2.1.5. Assignment Relation. As a project proceeds, eligibility rules are tested by accessing the underlying facts stored in a database and as historical individuals are found that meet the constraints they are actually assigned to be a given element of a process. This information cannot be captured in the eligibility relation because that would cause the eligibility rules to be lost in the event that the individual is later de-assigned (e.g., due to a resource conflict) and would also preclude reuse of the process description that lead to the current state of the project at a later time.

The basic requirement is to create a relation that keeps track of all the individuals that are available for assignment as well as their assignment status; either which element (or elements in the case of resource competition) they are assigned to or the fact that they are not currently assigned at all.

An obvious solution to this relation would be

(<u>INDIVIDUAL,PPSO,ELEMENT,STATUS</u>)

Note that, like it or not, the combination of PPSO and ELEMENT is a globally unique name and it could not be otherwise since you have to know to which element the assignment is being made [21]. If an individual is unassigned then the PPSO will be null or L1 depending on how one thinks about it. If an individual is pending assignment to several elements, then there will be multiple entries in the relation and the STATUS field will reflect the conflict. And it may well be the case that an individual <u>really is</u> assigned to more than one element (e.g., a single person is frequently both the Principal Investigator <u>and</u> the Project Manager). Hence, it really is necessary that all of the domains in the relation are needed to form the key. Otherwise, you would not be able to distinguish among the above possibilities (i.e., when multiple entries for an individual exist for different reasons).

4.2.1.6. Connectivity Relation. If all of the above relations exist for a given PPSO <u>except</u> the temporal constraint relation (it has no entries for that PPSO) then what we have is either an achievement description or a process description [22]. If we proceed to record the before and after state of affairs related to a process, then we need to capture the information relative to that connectivity. (Interestingly enough, it is precisely that information that represents the formal progress reporting of conventional scheduling systems). Thus we need something of the form

(BPPSO,<u>PPSO</u>,APPSO)

where PPSO is the process, and the other two are the before and after states of the project (achievements). It's probably not the case that you can get away with only PPSO as the key since you could have only one paradigm for the process yet several paradigms for the before and after states of affairs. The converse will not be the case since each paradigm for the process has a unique name. There is a legitimate question remaining when it is the case that the before and after achievements are the same, even though they're different paradigms; i.e., can you eliminate the paradigm information altogether [23].

4.2.1.7. Summary of the Normalization. The basic relations summarized in preceding paragraphs would appear to capture the information essential to both process and achievement descriptions, at least as the inference engine needs to see it. There will, undoubtedly, be additional relations for the benefit of the user (explanatory names, textual descriptions, etc.) although the inference engine will not require these even though the essential relations could be automatically derived from these in a later version. Many of the aforementioned problems were not evident in the reference problem but they all appear likely to occur in any reasonably complex schedule and it is thus better to deal with them now since they significantly complicate the inference engine design.

4.2.2. "Inference" Engine. An implementation of a Knowledge Dictionary System for scheduling support necessarily includes the functionality to create, update, and maintain the schedule description. This is already exists as a prototype system developed partly prior to and partly in connection with the investigation and is described in Section 3.3.1. The work discussed in this section is principally the development and augmentation of part-whole inferencing to satisfy the requirements of scheduling support.

4.2.2.1. Basic Part-Whole Inference. The software for the basic process of part-whole inference is similar to but more extensive than rule processing. The specific requirement to be satisfied by the basic part-whole inference engine is as follows. If W1 and W2 are schedules (which may be parts of other schedules, etc.); and W1 is known to have occurred if its stages, P1 and P2 have occurred; and W2 is known to have occurred if W1 has occurred; then basic part-whole inference is capable of inferring any or all of the following:

(a) if the elements of P1 and P2 have been satisfied (i.e., there exists at least one individual eligible to be each element), then P1 and P2 have occurred;

(b) if P1 and P2 have occurred then W1 has occurred;

(c) if W1 has occurred then W2 has occurred;

(d) if W1 has occurred, then the elements of P1 and P2 have been satisfied;

(e) if W1 has occurred, then P1 and P2 have occurred.

4.2.2.2. **Resource Competition.** The basic process of part-whole inference, alone, is insufficient for dealing with the resource competition problem posed by scheduling. It it necessary to have a specific algorithm for tracking resource competition, including the ability to provide notification when such competition exists. The algorithm is as follows. If P1 and P2 are both processes serving as options of S1 and S2 respectively and requiring resources R1 and R2 respectively; and R1 and R2 are both of type R; and R > P1 + R2; the part-whole software needs to:

(a) detect this as a partial resource conflict;

(b) note P1 and P2 as being in conflict because of R;

(c) provide the information in (b) to the user on request;

(d) find a set of combinations of other options for S1 and S2 {(Px,Py)} that do not conflict because of R;

(e) provide the information in (d) to the user on request;

(f) continue the analysis of P1 and P2 in terms of their requirements for other resources.

By virtue of this algorithm, the user may suspend the inference, substitute one of the (Px,Py), and restart the inference from the point of conflict.

4.2.2.3. Temporal Conflicts. The basic process of part-whole inference is also deficient for scheduling in that it lacks a specific algorithm for tracking temporal interference, including the ability to notify the analyst when such interference exists. Such an algorithm is defined as follows. If P1 and P2 are processes serving as options of S1 and S2 respectively, and P1 cannot begin until P2 begins (or occurs or ends, etc.); and P2 is a process with a similar temporal dependence on P1; and these dependencies are in conflict; then the software shall:

(a) detect this as a partial temporal conflict;

(b) note P1 and P2 as being in conflict because of their temporal state (beginning, occurring, ending, etc.);

(c) provide the information in (b) to the user on request;

(d) find a set of combinations of other options for S1 and S2 {(Px,Py)} that are not in temporal conflict;

(e) continue the analysis of P1 and P2 in terms of their other temporal dependencies.

By virtue of this algorithm, the user may suspend the inference, substitute one of the (Px,Py), and restart the inference from the point of conflict.

4.2.2.4. Choice Principles. The basic process for part-whole inference needs to be augmented with algorithms for both the automatic and manual application of choice principles. These algorithms shall accomplish the following. If P1 and P2 are processes serving as options for S1 and S2 respectively and P1 and P2 are competing for an insufficient resource R, and there exists a stored choice principle that states "anytime R is insufficient, the scenario that requires more of R is less desireable," and P2 requires more of R than P1, then the software will automatically:

(a) cease to consider P2 in terms of R;

(b) find an option for S2 (Px) that requires less (or none) of R;

(c) continue the analysis of S2 using Px as an option.

Similarly, if P1 and P2 are in temporal conflict and there exists a stored choice principle that states "any process that began before another process with which it is in temporal conflict is more desireable" and P2 began after P1 then the software will automatically:

(e) cease to consider P2 in terms of its temporal dependence on P1;

(f) find an option for S2 (Py) that is not in temporal conflict with P1 (or can begin earlier than P1);

(g) continue the analysis of S2 using Px as an option.

If no stored choice principles exist, the software will simply have to note the conflicts.

4.2.2.5. Propagation of Reports. The basic part-whole inference process needs to be augmented with specific algorithms to exploit intermixed description types. The algorithms shall accomplish the following. If Sa is the achievement that exists before process Pa begins; Sab is the achievement that exists after Pa ends but before process Pb begins; and Pa and Pb are the only stages of process P (i.e., P always occurs if Pa and Pb do), then the software shall be able to infer at least any or all of the following:

(a) if Sa does not exist then Pa cannot have begun;

(b) if Pa has begun then Sa does (or did) exist;

(c) if Sab exists then Pa has ended;

(d) if Pb has begun then Sab does (or did) exist;

(e) if Sb exists then Pb (and consequently Sab, Pa, and Sa) have all occurred;

(f) if Sb exists, then P has occurred;

(g) if P has occurred, then all of Sa, Pa, Sab, Pb, and Sb have all occurred.

4.3. Prototype System Overview.

The knowledge necessary to perform and analyze complex scheduling must eventually be stored and maintained in a structured tabular form for processing by the part-whole inference engine. The process of developing this knowledge base typically begins with narrative descriptions of the project; proceeds to more structured textual descriptions (e.g., outlines, pseudo-code); and concludes with an admixture of structured tables (for use by the inference engine) and related discursive explications (for the convenience of the user). Such a progression requires the support of a word processor, a text editor, and a database system. And because the discursive information persists even in the final knowledge base, all three are required concurrently and continually throughout the scheduling effort. While it may be possible to maintain the knowledge base in three separate systems with appropriate interfaces, our experience has been that a single system possessing the combined functionality of all three is clearly called for.

Building upon earlier work [24], a Knowledge Dictionary System was implemented for the purpose of creating, updating, maintaining and searching a scheduling knowledge base. This system combines, in a single integrated environment, functions of word processing, text editing, and database management. The functionality of this system is described in Appendix A of this report.

5. Scheduling Knowledge Base

The Phase I investigation was done with respect to an existing reference problem to provide an actual historic particular basis in addition to a conceptual basis for the investigation. The context of the reference problem is the scheduling of the UTAIN/MAIS effort, information about which was provided by the COTR. Within that context, it was decided that the reference problem would be the tracking of compatibility between the Operating System and the Network Software. This problem was chosen for several reasons: it was sufficiently complex to test the power of our approach; being purely technical, it did not require the use of classified information; and, most importantly, since none of the provided information was of sufficient depth, it is a subject with which we were sufficiently familiar to be able to generate our own data.

This section is divided into four parts: the first is a parametric analysis of the issues involved with Operating System (OS) software, the issues involved with Local Area Net (LAN) software, and their interrelationships. The second is the technical development of a compatibility matrix between the OS and LAN software (the core of the reference problem). The third is a scenario description (narrative) for that problem in the context of the larger social practice of scheduling. And the fourth is the structured Social Practice Description as implemented in the system.

5.1. OS/LAN Technical Issues. To understand the complexities that arise in the scheduling problem as a result of maintaining compatibility between the two major software systems and their interfaces, it is first necessary to understand something about each independently of the other. This was accomplished during the investigation by resorting to parametric analysis: i.e., identifying the major technical issues of OS and LAN software respectively and decomposing them into successively smaller issues until the point is reached that each issue is a "parameter" in that, by assigning it a value (either qualitative or quantitative), one characterizes the system of which it is a part as different in an important way from some other system that had a different value for that parameter. This parametric analysis is presented as Appendix B of this report.

5.2. OS/LAN Compatibility Matrix. The compatibility between the OS and the LAN software was developed as a matrix with the vertical axis corresponding to the OS parameters and the horizontal axis corresponding to the LAN parameters. Compatibility is stated at multiple levels of the part-whole hierarchy. Each statement is a relationship name in brackets "[nameK]" that names the relationship that the OS component must have with the LAN component. The matrix, due to its size, is presented in "strips" and pages. The strip number refers to the sequence, from left to right, that the page would be placed if the matrix were actually to be pasted together. The page number obviously refers to the depth. The notation ">>" in the horizontal axis indicates that the column is above the following column in the part-whole hierarchy. The part-whole depth in the vertical axis is represented by paragraph numbering and indenture. The matrix is provided as Appendix C of this report to illustrate the type of compatibility problems that must be dealt with in the course of scheduling.

The matrix is largely vacuous because it became apparent early in its development that the ability to process such matrices was critical to demonstrating feasibility. Hence, a compatibility matrix inference engine was developed.

5.3. Structured Social Practice Description

As discussed earlier, a Social Practice Description is a formal definition represented as tables. The social practice of scheduling a large and complex local area network is presented in Appendix D and is comprised of the Social Practice Description (SPD) table, and the Element--Individual List (EIL).

5.4. Scheduling Scenario Description

What transforms a social practice into a scenario are the actual facts (historical particulars) about an occurrence of the social practice. The scenario used in the investigation deals with the construction of a distributed data handling system (DHS). The scenario involves a prime contractor (TRW), a subcontractor for the hardware (IBM) and a subcontractor for the network software (DEC). This framework provides the context for two major types of contingencies: (a) delays in the installation of the hardware lead to eventual delays in the installation of the software after several revised completion estimates are generated; and (b) opportunities for mismatches between operating system and network components require systematic analysis grounded in the compatibility matrix. The scenario is formally presented in Appendix E as a Fact Table and a Fact Type Table.

6. Future Research and Development.

The principal result of the Phase I effort was the design of and confirmation of the practicality of developing a Knowledge Dictionary System for scheduling support. The immediate goal is to actually implement that prototype as an extension of the implementation used and partly developed in Phase I. A stand-alone prototype for a Knowledge Dictionary System capable of managing the knowledge necessary to perform and analyze complex scheduling would provide a foundation for both: (a) future research; extending the problem domain beyond that of scheduling; and (b) future development; engineering a maintainable and extensible product that can be integrated with existing workstation, network and mainframe environments.

6.1. Problem Domain Extensibility.

One important direction of future research is to exploit the inherent generality of the SA technology upon which the Knowledge Dictionary System is built; and to demonstrate its applicability to problems other than scheduling.

6.1.1. Decision Aid Framework.

Scheduling, while a significant problem in itself, is, on a larger scale, merely an exemplar of the class of problems characterized by the need to discriminate between what is actually happening and that which was planned (or expected, or desired, etc.). The two paradigmatic exemplars are: (a) control; altering what is actually happening based upon what is desired; and (b) forecasting; altering what is expected based upon what is actually happening. Moreover, all of these fit the even more general

value -> action -> value

framework of all human decision making. That is, actions are undertaken in order to transform an existing state of affairs into a new one of a desireable sort; or in order to prevent the state of affairs *from changing* into a *new* one of an undesirable sort. State of Affairs (SA) Technology, upon which the Phase II KDS is modelled [25] [26] [27], effectively represents this framework by recognizing that decision making is concurrently both completely situation--dependent and completely principled [28].

6.1.2. Situation Dependencies. The choice of an appropriate action requires at least: (a) knowledge of "this" state of affairs; (b) knowledge of actions actually available in "this" state of affairs including limitations imposed by limited knowledge; and (c) knowledge of the value (desirability) of the "this" state of affairs and the consequent state of affairs if any of the actions actually available are taken. Because of this, deductive schemas for going directly from observable facts to observable actions to observable consequences are not available as rigorous methods [29]. The distinction in SA between the elements (abstract place-holders) of a process, and the historical, particular individuals eligible to be those elements in a given "instance" of the process (the assignment relation) effectively models situation dependencies.

6.1.3. Choice Principles. By contrast, it is also (and concurrently) the case that choices are not arbitrary. Instead, they exhibit the inherent rationality of decision making as exemplified by abstract, context--free maxims [30] such as: (a) if a person wants to do something, he has a reason to do it; (b) if a person has two reasons for doing something, he has a stronger reason to do it than if he had only one of those reasons (c) if a person has a reason to do something, he will do it unless he has a stronger reason to do something else instead; (d) if the situation calls for a person to do something he cannot do, he will do something he can do. These are all exemplars of the completely general principle that "A person values some states of affairs over others and acts accordingly" (the value -> action -> value framework noted above). As such, while SA descriptions do not prescribe which actual choices to make in a given situation, they do operate as a logical constraint on the possibilities of choices and thus effectively model the rationality of choice making.

In summary, the goal of future research is to exploit the general decision making framework inherent in the Knowledge Dictionary System and to demonstrate its applicability control applications and forecasting applications.

6.2. Product Engineering.

A prototype Knowledge Dictionary System for scheduling support would provide a strong foundation for a maintainable and extensible product that can be integrated with existing workstation, network and mainframe environments. This effort subdivides into two parallel but distinct efforts: (a) software engineering; transitioning the prototype, stand-alone software to a level of quality consistent with that of viable commercial products; and (b) knowledge engineering; developing archetypal knowledge bases for the major scheduling paradigms found in those Government and commercial enterprises targeted as potential customers.

6.2.1. Software Engineering. Given the anticipated functionality of a prototype Knowledge Dictionary System for scheduling support, we expect there to be a significant potential marketplace for a product of this kind. But regardless of who the prospective customers may be (these are discussed in the Section entitled Potential Post Applications) the differences between a prototype and a commercially viable product are significant in several ways. As opposed to a prototype that is designed specifically to demonstrate functionality, a commercial product is designed to be: (a) supportable; the software must be highly modular, well documented, and subjected to rigorous systematic version and release control; (b) integrative; the software must yield easily to coexistence with the dominant operating system and database environments in the marketplace; and literate; i.e., there must be a combination of user-oriented documentation and user-friendliness sufficient to overcome the initial static friction so that product acceptance can be obtained [31].

6.2.1.1. Support. The system used and augmented during the Phase I investigation is already highly modular and possesses a well-defined protocol for implementing additional functions. It is highly amenable to being subject to source code control since it is written in Borland Pascal Version 4 which provides full support for Units and for separate compilation. The principal effort in this regard is the documentation of the source code (notably sparse in the existing system).

6.2.1.2. **Integration.** As written, the existing system and, hence, any prototype derived from it, is well-behaved under MS-DOS 3.x [32] and should run in any upward compatible environment such as OS/2 and any such environment running as a task under another environment (e.g., Unix, Novelle, Locus, etc.). To the extent that the new environment provides additional services that must be requested (e.g., lock management) the modification to access these services system is minimal. The existing system contains its own file service through which <u>all</u> I/O requests of the command modules are processed. Finally, and most importantly, the <u>entire</u> knowledge base is represented as flat tables. By virtue of this, and in combination with the internal file service, modifying the system to utilize parts (or even all) of the knowledge base stored externally in a relational database system is very straight forward.

6.2.1.3. **Literacy.** By far, the most dominant effort in developing a product will be the preparation of user manuals, tutorials and on-screen help as well as the refinement (or possibly redesign) of the user interface and the addition of extra-system functions (i.e., access to operating system functions from within the KDS).

6.2.2. Knowledge Engineering. As already discussed as part of the technology that enables the KDS to be effective, processes have paradigms (major variations). This fact is no less true for the process of scheduling, itself. Different organizations perform and analyze complex scheduling in paradigmatically different ways. And even within an organization, scheduling may be performed in different ways according to the task. For example, a defense contractor preparing a proposal is actually scheduling (a) the proposal effort; (b) the effort being proposed; and (c) the actual effort if the proposal is successful. Each of these will almost always be done by different individuals, and very possibly in different ways. Just as general purpose accounting systems are often delivered with a set of pre-defined or partially pre-defined charts of accounts for different types of businesses (i.e., give the user someplace to start); so do we envision that the Phase III system will be delivered with a set of partially pre-defined knowledge bases, each corresponding to one of the major scheduling paradigms. This, too, will give the user someplace to start; i.e., for each project with which he is confronted, he will be able to copy the most appropriate paradigm and then particularize it to the specific project. The effort to accomplish this requires: (a) paradigm formulation; the analysis that leads to the selection of the scheduling paradigms; and (b) paradigm description; the development of representation units and their elements that correspond to each paradigm.

6.2.2.1. Paradigm Formulation. Different paradigms of scheduling are not merely arbitrary variations of scheduling, but are, instead, fundamentally different versions that are selected by a structured approach. Typically, the fundamental difference that calls for another paradigm is that the process in one version has a different decomposition than in another (i.e., the parts and the part-whole relations are different) making it exceedingly complex to describe merely in terms of co-occurrence constraints. Discovering paradigms will be accomplished by Paradigm Case Formulation (PCF) [33] [34]. The steps in a PCF for scheduling would be to: (a) select an example of a scheduling method (this is actually accomplished as a result of Phase II in that the reference problem of a large DoD project will serve as the exemplar) [35]; (b) hypothesize transformations of the parts and part-whole relationships inherent in the paradigm case; (c) permute the paradigm case by inducing the transformations; and (d) select those permutations that are, themselves, both genuine exemplars of scheduling and correspond to (or are applicable to [36]) enterprises in the marketplace.

6.2.2.2. **Paradigm Description.** For each selected scheduling paradigm, the knowledge necessary to perform and manage that paradigm will be developed and stored in the Knowledge Dictionary System as a paradigm at the level of L1, and descending only to levels below which it would be necessary to particularize the description to a given project. The means for constructing the knowledge base have already been discussed at length elsewhere in this proposal. An abbreviated example of the process descriptions for a scheduling paradigm appears in Table 3 to illustrate how the depth of description can be limited to make the paradigm usable for a class of projects.

TABLE 3 - SCHEDULE PARADIGM (BPU)

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	TS
1	0	0		0	*	*	<Obtain>	*
1	0	0		0	*	*	[User]	*
1	0	0		0	*	*	[Approval]	*
1	0	0		0	*	*	[System]	*
2	0	0		0	*	*	<Execute>	*
2	0	0		0	*	*	[Contracts]	*
2	0	0		0	*	*	[System]	*
2	0	1		0	*	*	<Decompose>	*
2	0	1		0	*	*	[System]	*
2	0	1		0	*	*	[Subsystems]	*
2	0	2		0	*	*	<Develop>	*
2	0	2		0	*	*	[Subsystems]	*
2	0	2		0	*	*	[Integration]	*
2	0	3		0	100	*	<Procure>	*
2	0	3		0	*	*	[Contracts]	*
2	0	3		0	*	*	[Subsystems]	*
3	0	0		0	*	*	<Administer>	*
3	0	0		0	*	*	[Contracts]	*
3	0	0		0	*	*	[Subsystems]	*
3	0	1		0	*	*	<Review>	*
3	0	1		0	*	*	[Requirements]	*
3	0	1		0	*	*	[Subsystem]	*
3	0	2		0	*	*	<Review>	*
3	0	2		0	*	*	[Concepts & Facilities]	*
3	0	2		0	*	*	[Subsystem]	*
3	0	3		0	*	*	<Review>	*
3	0	3		0	*	*	[Preliminary design]	*
3	0	3		0	*	*	[Subsystem]	*
3	0	4		0	*	*	<Review>	*
3	0	4		0	*	*	[Detailed design]	*
3	0	4		0	*	*	[Subsystem]	*
3	0	5		0	*	*	<Review>	*
3	0	5		0	*	*	[Test readiness]	*
3	0	5		0	*	*	[Subsystem]	*
4	0	0		0	*	*	<Accept>	*
4	0	0		0	*	*	[Subsystems]	*
4	0	0		0	*	*	[Contractors]	*
5	0	0		0	*	*	<Integrate>	*
5	0	0		0	*	*	[Subsystems]	*
5	0	0		0	*	*	[System]	*
6	0	0		0	*	*	<Deliver>	*
6	0	0		0	*	*	[System]	*
6	0	0		0	*	*	[User]	*
7	0	0		0	*	*	<Maintain>	*
7	0	0		0	*	*	[System]	*
7	0	0		0	*	*	[User]	*
100	0	0		0	*	*	<Procure>	*
100	0	0		0	*	*	[Contract]	*

TABLE 3 - SCHEDULE PARADIGM (BPU) CONTINUED

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	TS
100	0	0		0	*	*	[Task]	*
100	0	1		0	200	*	<Develop>	*
100	0	1		0	*	*	[RFP]	*
100	0	1		0	*	*	[Task]	*
100	0	2	A	0	*	*	<Submit>	*
100	0	2	A	0	*	*	[RFP]	*
100	0	2	A	0	*	*	[Procurement]	*
100	0	2	B	0	*	*	<Submit>	*
100	0	2	B	0	*	*	[DD254]	*
100	0	2	B	0	*	*	[Billet requests]	*
100	0	2	B	0	*	*	[Security]	*
100	0	3		0	*	*	<Evaluate>	*
100	0	3		0	*	*	[Proposals]	*
100	0	3		0	*	*	[Task]	*
100	0	4		0	*	*	<Recommend>	*
100	0	4		0	*	*	[Proposal]	*
100	0	4		0	*	*	[Procurement]	*
100	0	5		0	*	*	<Assist>	*
100	0	5		0	*	*	[Procurement]	*
100	0	5		0	*	*	[Contract]	*
100	0	5		0	*	*	[Negotiation]	*
100	0	6		0	*	*	<Accept>	*
100	0	6		0	*	*	[Contract]	*
100	0	6		0	*	*	[Procurement]	*
200	0	0		0	*	*	<Develop>	*
200	0	0		0	*	*	[RFP]	*
200	0	0		0	*	*	[Task]	*
200	0	1		0	*	*	<Prepare>	*
200	0	1		0	*	*	[SOW]	*
200	0	1		0	*	*	[Task]	*
200	0	2		0	*	*	<Prepare>	*
200	0	2		0	*	*	[Schedule]	*
200	0	2		0	*	*	[Task]	*
200	0	3		0	*	*	<Prepare>	*
200	0	3		0	*	*	[Technical criteria]	*
200	0	3		0	*	*	[Task]	*
200	0	4		0	*	*	<Prepare>	*
200	0	4		0	*	*	[Budget]	*
200	0	4		0	*	*	[Task]	*
200	0	5		0	*	*	<Prepare>	*
200	0	5		0	*	*	[DD254]	*
200	0	5		0	*	*	[Task]	*
200	0	6		0	*	*	<Prepare>	*
200	0	6		0	*	*	[Bidders list]	*
200	0	6		0	*	*	[Task]	*
200	0	7		0	*	*	<Prepare>	*
200	0	7		0	*	*	[Billet requests]	*
200	0	7		0	*	*	[Task]	*
200	0	7		0	*	*	[Bidder]	*

7. Potential Applications.

The principal result of the Phase I is a specification for a fully functional, stand-alone prototype for a Knowledge Dictionary System capable of managing the knowledge necessary to perform and analyze complex scheduling associated with large DoD projects. This result is immediately applicable in both DoD agencies and in commercial organizations that engage in DoD contracting. Moreover, to the extent that civil agencies of the Government employ a similar scheduling paradigm, there would be the potential for additional applications throughout the Government and in private organizations that engage in Government contracting. (NASA would clearly be of primary potential in this regard.) Finally, as a result of future research and development, the Knowledge Dictionary System would have potential applications in both the Government and the private sector; first, to scheduling problems employing a wide variety of paradigms; secondly, to problems other than scheduling, particularly control and forecasting problems; and finally, as an integral part of large-scale data handling systems directed at these problems that, to some extent, are already in existence and have significant inertia in both the operating and database environments.

7.1. Commercial Post Applications.

The Knowledge Dictionary System has the potential to be commercially viable, both as a product and as a service, in any private enterprise that engages the problems of scheduling, control and forecasting at various levels of significance.

7.1.1. Immediate Potential. The immediate commercial value of the KDS is as a software product for scheduling complex, DoD-like projects that would operate in any PC or compatible environment. The immediate customer base would be organizations that perform this scheduling as an ancillary function to their principal line of business. This would require the establishment of both the marketing and support functions either internally or through the organization funding the Phase III effort. For small businesses or consultants, marketing would be accomplished primarily through advertising and support would be of the "hot-line" variety. For large DoD contractors, marketing would be accomplished by on-site seminars and demonstrations, and support would likely be provided as a separately contracted service to develop the complete knowledge base on a project by project basis.

7.1.2. Future Potential. As the problem domain expands during future research efforts, the base of potential customers would expand to include organizations that perform scheduling, control and forecasting. Most importantly, due to the increased sophistication of the product, the base would now include organizations that perform these functions as their principal line of business (e.g., financial planning, market research, investment brokers, etc.). This latter group is of particular interest in that they are not already consumers of commercial software and thus represent a market segment in which there is yet little or no competition [37]. For large organizations of this type, there is an added opportunity to market the KDS in concert with ongoing consulting services to develop and/or evaluate new paradigms in the principal line of business. This is equivalent to obtaining significant amounts of private funding from a variety of commercial sources to support an ongoing research program. Moreover, as the results of this research have very close parallels to intelligence analysis (see below), this, in effect, is an unusual case whereby the private sector would be funding the development of technology for DoD.

7.2. Government Applications.

The Knowledge Dictionary System also has the potential to serve the needs of the Government, and in particular, DoD. Besides offering to the Government the same potential applications already noted as commercial, in the DoD environment there are additional applications, both immediate and future, that on the surface appear to be quite different but, due to the SA technology that underlies the KDS are, in fact, fundamentally similar.

The similarity lies in the fact that the KDS is not inherently limited to just one world-view. For scheduling, only one world-view is called for and the KDS compares reported activity and/or achievements with the world-view to infer the disparity between what was planned and what is happening. For intelligence analysis, multiple world-views are called for and the KDS compares reported activity with each world-view to infer which is most consistent with what is happening. I.e., the world-views serve as possible scenarios as envisioned by the intelligence analyst; and the KDS serves to notify him of which scenario appears to be unfolding or, alternatively, that an anomalous scenario unforseen by the analyst is unfolding and requires explanation. This application is immediately applicable in any PC compatible environment (e.g., the Zenith 248 Local Area Network) and, as a result of product engineering, could be integrated with larger systems already supporting substantial databases.

8. References

[1]. McCarthy, J., "Generality in Artificial Intelligence," Communications of the ACM, Vol. 30, No. 12, (December 1987) pp. 1030-1035.

[2]. Ossorio, P.G., Schneider, L.S., Proposal for a Knowledge Dictionary System: Phase I, submitted to the Small Business Innovation Research Program, Rome Air Development Center, Griffiss AFB, NY, 1986.

[3]. Ossorio, P.G., "What Actually Happens": The Representation of Real World Phenomena, University of South Carolina Press, Columbia, SC, 1978.

[4]. PERT-based can and has been elaborated upon to deal with additional complexities but is forever limited by its foundation-level representation of a project. The fundamental concepts of representation that limit its ability to describe schedule complexities include:

(a) a project can be represented as a two-dimensional network of activities interrelated by time and resource requirements;

(b) an achievement (milestone) within a project can only be attained by completing the activities in the network that precede it;

(c) the way in which one achievement is attained is independent of the way in which other achievements are attained;

(d) an activity invariably consumes specific resources in known quantities;

(e) an activity invariably has other, specific activities as prerequisites;

(f) two (or more) activities that conflict in terms of a given resource can only proceed if an additional quantity of the resource is allocated;

(g) two (or more) activities that conflict temporally can only proceed if one or more of the activities are re-scheduled;

(h) progress is the attainment of an achievement.

[5]. For example, the report that a unit test was successful may be made be made retroactively inaccurate at a high frequency during integration testing while the acceptance of a deliverable does not need to be reconfirmed very often. When such is the case, inference based on simple backtracking (confirming all the lowest level milestones, then all next level milestones, etc.) is so impractical that, in complex schedules, either the inference will never be complete or will be hopelessly outdated when it does.

[6]. For example, a report may indicate that all the billets have been acquired and all the necessary contractor personnel have been briefed, but the number of available billets is insufficient for the number of contractor personnel.

[7]. For example, a schedule may require that the design of a message passing protocol for a network be started long before the design of the lower-level but less complex token passing protocol but that both designs are complete before the Preliminary Design Review (PDR); and that the design of the application datagram formats, while having a temporal relationship with the other two for commencement, have no such relationship for completion since they are not a part of the PDR.

[8]. For example, one report about the project might be that the network is capable of managing distributed transactions (a state of affairs report). Another might be that the distributed transaction management software has been successfully tested (an achievement report). And yet another might be that the operating system is distributing transactions among multiple servers (a process observation). Although all three reports are of a different character, and may well have come from different sources at different times, they are all reports of essentially the same transformation of the project.

[9]. Using the prior example, the development of the distributed transaction management software might obviously be inferred to be a part of whole task of developing the operating system. But if the operating system calls for distributed transaction management as a requirement, then it must also be inferable that if the operating system is

developed, distributed transaction management will be a part of that; and that achievement may well be a part of some other schedule such as the development of the database system.

[10]. In fact, a major design goal of the prototype which was successfully achieved was that <u>all</u> accesses to the State of Affairs structure are "piped" through <u>one common procedure</u>. While a degree of efficiency is sacrificed by this architecture, the advantages cannot be overstated. Integrating the State of Affairs System into either an environment in which the State of Affairs Structure is maintained in a centralized mainframe database; or one in which that structure is maintained on one or more servers in a distributed network; would require modification of less than one percent of the existing code, and that would encompass no more that a straight forward engineering modification.

[11]. The representation of date-time groups in the reasoning chain is essential to the efficiency of the process (it would literally be impossible to continually attempt to reconstruct everything from the atomic facts) and represents the fact that some reports are highly durable while others are very transient.

[12]. The question of "generalization" or "inheritance" becomes very important at this point. Will it be the case that, for example, an eligibility stated at L1 will automatically apply to any sub-context? Given the issues of mere description versus classification versus appraisal such a decision requires some thought. Note that the person maintaining the knowledge base and the person using to draw conclusions may be different and one may be using the generalization property to mean something different than the other and this could lead to serious misinterpretations.

[13]. One can argue that while individuals do, in fact, require global names for dealing with resource competition, elements do not require a name outside the context of the process in which they occur. However, in order to make an historical assignment of an individual to an element, you have to know in which process the assignment occurs. Thus every element name will have to be tagged with the FPSO name which necessarily makes element names unique. This will become apparent in the discussion of the assignment relation.

[14]. However, if co-occurrence constraints and eligibilities become intermingled (e.g., a given eligibility only applies if a given co-occurrence constraint is satisfied) then a more complex structure for all eligibility constraints needs consideration. A general PROLOG-like rule structure would probably work (and would be feasible given that Borland Pascal Version 4.0 delivers on it's promise of PASCAL-PROLOG linkable object modules) but would incur the traditional inefficiencies of that approach.

[15]. There are two objectives in conflict here. Literally speaking, the "stages" of a process are sequential and, in a schedule, a lot of attention is focused on stages because that is where, in the process, that things come together (perhaps this is a case of looking under the lamppost for the lost key given the present limitations of scheduling systems). But within a stage there is a great deal of temporal complexity which, if not dealt with explicitly, will cause conflicts due to cross-constraints to be lost in the shuffle (i.e., they will appear simply as two or more stages in different processes that are behind schedule without any recognition that the reason they are behind is that they are in competition with each other).

[16]. Gray, J.N., "Notes on Database Operating Systems", in <u>Operating Systems: An Advanced Course</u>, vol. 60, Springer--Verlag, 1978, 393-481.

[17]. Note that it may also be the case that stages of a process don't have any temporal dependence. E.g., both stages have to occur in order for the process to occur but either can occur at any time independently of the other. Depending on the chosen solution, this may simply be a null case that does not require any further analysis.

[18]. And herein lies the hope that the process can be recursive as previously discussed, although this may only be the case if we do, in fact, view it as a queuing problem.

[19]. Of course, there must be some stages that have no dependencies or the process can never start; and there can't be any cycles in the description of the dependencies or the process will simply stall (although dynamic cycles can occur but more on that later).

[20]. Ossorio, P.G., Place, LRI Report No. 30a, LRI, Inc., Boulder CO, 1982.

[21]. At this point, the question really becomes one for the user. If he finds it useful to develop a systematic element naming system in L1, there's nothing to prevent his doing that. If the element names are arbitrary except in the context of a process, then that's fine too.

[22]. This is why it was probably a good idea to model on the process description since it can get considerably more complex than the achievement description.

[23]. Strictly speaking, we take it to be the case that any paradigm of a process always has the same before and after states of affairs; i.e., They are different versions of the same process. On the other hand, it is often the case that the reason for having two paradigms is that one will work under a given set of conditions (i.e., state of affairs) while another will not. Until we achieve further resolution of this issue, we use the term "paradigm" to denote any version of a process that has the same elements although with possibly different eligibilities. Versions with different elements are considered to be different processes. The intent is to focus on different decompositions of the same process.

[24]. Ossorio, P.G., Schneider, L.S., Final Technical Report, Contract F-30602-85-C-0190, Rome Air Development Center, Griffiss AFB, NY, 1987.

[25]. More precisely, the Phase I KDS is modelled on a restricted subset of SA that deals only with the concepts of process and achievement, i.e., those concepts essential to the problem domain of scheduling (actually, the subset of that problem domain that we chose as the reference problem of the Phase I investigation). In a system modelled on an unrestricted SA the articulation of the concept of reality is accomplished by reference to the four basic constituents, namely, "object," "process," "event," and "state of affairs," and their further development. Note that these are not invented technical terms. Rather, they are already straight-forwardly concepts of reality or the real world. A primary and paradigmatic use of these concepts is as the categories of "what there is." Also, and by no means unrelated, the four concepts are observation concepts - we observe exemplars of each kind. The fact that our separate

observations can be formulated as observations of a single world; i.e., the real world, requires that there be logical relationships among the concepts in terms of which our observations are made and our world described. These are expressed as a set of transition rules.

[26]. The transition rules for an unrestricted SA system are:

T1) A state of affairs is a totality of related objects and/or processes and/or events and/or states of affairs.

T2) A process (or object or event or state of affairs) is a state of affairs which is a constituent of some other state of affairs.

T3) An object is a state of affairs having other, related objects as immediate constituents (an object divides into smaller, related objects).

T4) A process is a sequential change from one state of affairs to another.

T5) A process is a state of affairs having other, related processes as immediate constituents (a process divides into related, sequential or parallel, smaller processes.

T6) An event is a direct change from one state of affairs to another.

T7) An event is a state of affairs having two states of affairs as constituents (i.e., "before" and "after").

T8) That a given state of affairs has a given relationship to a second state of affairs is a state of affairs.

T9) That a given object, process, event, or state of affairs is of a given kind is a state of affairs.

T10) That an object or process begins is an event and that it ends is a different event.

T11) That an object or process occurs (begins and ends) is a state of affairs having three states of affairs as constituents (i.e., "before," "during," and "after").

[27]. The limiting cases for an unrestricted SA system are:

L1) The state of affairs which includes all other states of affairs (i.e., a world view).

L2) An object that has no constituents, hence is an ultimate particle (i.e., a stubbed-off object definition).

L3) A process that has no constituents, hence no beginning that is distinct from its end (i.e., an event).

L4) An event that has no constituents, hence the equivalent of an object during a period during which the object undergoes no change (i.e., a timeless state of affairs).

[28]. Ossorio, P.G., Schneider, L.S., Decisions and Decision Aids, LRI Report No. 31, Linguistic Research Institute, Boulder CO, 1982.

[29]. To be sure, we sometimes employ logical or mathematical algorithms as definitions of actions, but only after the prior decision has been made that in "this" situation (or this kind) such a schema is relevant. Blindly following a formula which says "Whenever X do Y" where X and Y are concrete descriptions of facts and actions (e.g., Whenever you're outnumbered, retreat.) is a prescription for disaster, for such a formula will have genuine value only under extremely limited conditions and cannot provide a general basis for decision making.

[30]. Op. cit. Place.

[31]. This is, perhaps, the most critical issue to be addressed. Notably successful products have, in general, been introduced with a plethora of manuals, tutorials, menu-driven interface options, and on-screen help functions, most of which are ignored by the user soon after installation; but without which the user would never have even attempted to try the product. And this is especially true for extremely powerful systems due to their necessarily inherent complexity. WordPerfect (SSI Software), the most widely used word processing software in PC environments, is a paradigmatic case in point. By contrast, TK!Solver

(Software Arts, Inc.) may be the most powerful spreadsheet program ever developed but was notorious as a product outside of a very select community of engineers.

[32]. The only exception to this is screen I/O, which is discussed in the Section entitled Installation, and which would only be potentially problematic at the level of individual workstations.

[33]. Ossorio, P.G., "Conceptual-Notational Devices," in Davis, K.E. (Ed.), Advances in Descriptive Psychology (Vol. 1, pp. 83-104), JAI Press, Greenwich, CT, 1981.

[34]. The formal definition of PCF is inductive as follows.

PCF_1 ::= I. Introduce a Paradigm Case of X.
II. Introduce transformations of the Paradigm Case.

PCF ::= I. PCF_1.
II. T1. The number of Paradigm Cases are $K > 1$.
T2. The Paradigm Case is a generator of X.
T3. The Transformation of a Paradigm Case is a Paradigm Case.
T4. A Transformation may be any functional equivalent thereof.

[35]. While the choice of a paradigm case is conceptually arbitrary, in practice it will make a difference and sometimes a crucial difference. The choice of the reference problem as the paradigm case was made in accordance with the following rules of thumb developed through experience with PCF. The first is that the paradigm case be the most complex case so that the transformations are also simplifications. The second is that the paradigm case be indubitable. I.e., if ever there were a case of a scheduling problem, that's one! The final rule is that the paradigm case be in some relevant sense a primary or archetypal case. This gives formal recognition to the fact that the other cases generated are cases because of their relation via the transformations to the paradigm case; and, hence the transformations serve as explanations of why the paradigms are different.

[36]. The effort is not an investigation to discover new ways to perform scheduling. However, by virtue of using PCF, the effort potentially has this benefit as a side-effect. It is possible that PCF will yield a scheduling paradigm that, while exhibiting very desireable characteristics and applicability, does not appear to be in use by anyone. PCF is analogous to a Zwicki Morphological Analysis in this regard.

[37]. While they may employ commercially available systems to support these as internal functions ancillary to their line of business, whatever software they use in support of their product (e.g., a stock market forecast) is typically developed in-house and preciously guarded. Since the KDS is particularized by the content of the knowledge base, not by the software, such organizations could obtain the many benefits of KDS without compromising proprietary knowledge.

A Knowledge Dictionary System for Scheduling Support

P.G. Ossorio and L.S. Schneider

Appendix A

Submitted by
Linguistic Research Institute, Inc.
5600 Arapahoe Avenue
Boulder, Colorado 80303

Submitted to
Rome Air Development Center
Griffiss AFB, New York

TABLE OF CONTENTS

LIST OF EXHIBITS

1. Prototype System Overview.

The knowledge necessary to perform and analyze complex scheduling must eventually be stored and maintained in a structured tabular form for processing by the part-whole inference engine. The process of developing this knowledge base typically begins with narrative descriptions of the project; proceeds to more structured textual descriptions (e.g., outlines, pseudo-code); and concludes with an admixture of structured tables (for use by the inference engine) and related discursive explications (for the convenience of the user). Such a progression requires the support of a word processor, a text editor, and a database system. And because the discursive information persists even in the final knowledge base, all three are required concurrently and continually throughout the scheduling effort. While it may be possible to maintain the knowledge base in three separate systems with appropriate interfaces, our experience has been that a single system possessing the combined functionality of all three is clearly called for.

Building upon earlier work [1], a Knowledge Dictionary System was implemented for the purpose of creating, updating, maintaining and searching a scheduling knowledge base. This system combines, in a single integrated environment, functions of word processing, text editing, and database management. The functionality of this system is as follows.

2. User Interface Environment.

The user is typically presented with a screen such as illustrated in Figure 1. The first line is referred to throughout as either the Status Line or Command Line. Underneath the Command Line is the Menu Bar, and beneath that are two windows of data. The Status Line conveys information about the currently active window. In the illustration, this indicates that Window 1 is opened to a file called KSC\DATA\INSTAL which is 255 characters wide and contains 124 rows (the remainder of the information on the Status Line is explained subsequently). The Menu Bar is an array of headings, each of which correspond to an object upon which the system can operate. These objects are Scans, Queries, Transactions, Windows, Files, and Relations. The functions of which the system is capable can be accessed in any of three ways.

2.1. Menu Bar. The menu bar represents the major functions that can be executed by the user. The menu bar is activated by F10 and the submenus are selected either by positioning with the arrow keys and pressing CR or by typing the capitalized letter of the selection. ESC backs out of any menu selection and a second ESC deactivates the menu bar. For example, typing F10, moving the cursor across the File heading, pressing CR, moving the cursor down to the

Quit selection and pressing CR again will yield a prompt on the Command line "^KX Exit (Y/N)?" Typing a "y" will exit the system. The same function can be accessed by typing F10, F (for File), Q (for quit). Not all of the available functions are accessible from the menu system. The submenus are illustrated in Figure 2.

2.2. Control Keys. Every available function of the system is accessible via special keys (e.g., arrow keys, tab, etc.) and one or two key sequences of control characters. In the above example, note that the prompt was preceded by "^KX" which is the control key sequence for the Quit function. I.e., holding down the Ctrl key, type kx and the same prompt will appear.

2.3. Macro Commands. Any combination of commands and responses to prompts can be invoked from the keyboard. A Macro Command is an ASCII file of a sequence of keystrokes and referred to by the name of the file in which the keystrokes are stored. A Macro Command is executed by typing F9 (which returns the prompt "Execute File:") followed by the file name and CR. This will cause the system to respond sequentially to each character in the file as if it had been typed from the keyboard.

3. Files.

A file is a data stream which can be viewed or modified through a window. Files are of three types: Tables, Text (a table with one column), and Documents (a "table" with no columns). To both the system and the user all types are pretty much the same except some functions behave differently for each type and not all functions are applicable to all types. The operations that can be performed with files are as follows.

3.1. Open File. A file can be OPENed in a window. This makes the file visible to the user and available for searching or updating.

3.2. Close File. A file can be CLOSEd, i.e., made invisible and unavailable for searching or updating.

3.3. Read File. Internally, files are maintained in a unique structure to facilitate the various operations of the system. But if the user has a standard ASCII file (e.g., produced by another program) it can be read in its entirety into the system and converted.

3.4. Write File. A user can also WRITE an entire file in standard ASCII format.

3.5. Top File. This positions the cursor to the first character of the first row of the file.

3.6. Bottom File. This positions the cursor to the last character of the last row of the file.

3.7. Synchronize File. Unlike typical word processors or text editors, the system does not read the entire file into RAM for processing. It, instead, employs a caching scheme whereby only the data most needed is resident in memory. Unless the user takes advantage of Transaction Management (see below) it is advisable to periodically SYNCHronize the file; i.e., force all changes to be recorded on the disk; just as one would periodically save a file when using a text editor.

3.8. Quit. This is a quick shutdown function to synchronize and close all opened files and exit to the operating system.

4. Windows.

Windows are viewing areas on the screen consisting of 1 or more display lines. New or empty windows are attached to temporary files called TEMP1, TEMP2, etc. and will remain so until the window or file is closed (if the user wants to save a temporary file, the system will prompt him for a permanent name). The operations that can be performed with windows are as follows.

4.1. Select Window. To operate on data in a window it must be SELECTed, i.e., made the current window. The cursor appears in only the current window and the Status Line always refers to the current window.

4.2. Open Window. A new window can be OPENed, i.e., made to appear on the screen. The user must always specify how many rows the window will contain (its length) and which existing window it will overlay. A window can be opened directly to a file or it can be opened empty in which case the user must specify how wide (in characters) the window should be.

4.3. Close Window. An existing window can be CLOSEd (removed from the screen). If the window is opened to a user file, the file will be closed automatically. If the window is opened to a non-empty temporary file, the user will be prompted for a file name if he wants that file saved.

4.4. Clear Window. Clearing a window is identical to closing it except that the window will remain on the screen attached to an empty temporary file.

4.5. Link Window. A window can be linked to the same file to which an existing window is already opened for the purpose of having two or more areas of the file visible at the same time. Changes made to the file from any of the windows will appear in all of the other windows (immed- iately, if the viewing area overlaps).

5. Transactions.

Changes to the data may, at the users option, be governed by a transaction management protocol. A transaction is a sequence of changes that are: atomic; i.e., either they all occur or none of them do and durable; i.e., once made, they can only be changed by another transaction.

5.1. Begin Transaction. BEGIN starts a transaction. It has the effect of establishing an opening parentheses in an equation in that everything within the parentheses will be treated as an atomic unit. When a transaction is begun, the status line will be augmented to indicate the number of rows that have been changed since the transaction began (Log) and the remaining capacity of the log, in rows, to absorb further changes (once a row has been changed, further changes to it do not require additional log capacity).

5.2. Commit Transaction. COMMIT ends a transaction favorably; i.e., it has the effect of establishing the closing parentheses and then performing all the changes that occurred within the parentheses by applying the changes to the disk.

5.3. Abort Transaction. ABORT ends a transaction unfavorably; i.e., it has the effect of establishing the closing parentheses and then undoing all the changes that occurred within the parentheses by restoring the disk to the way it was before the transaction began.

5.4. Enter Transaction. If a transaction has begun in one window, the user may desire to have changes in another window be a part of that transaction (i.e., they will have the same destiny as that transaction when it ends). This is accomplished by entering the transaction.

5.5. Leave Transaction. Once a transaction has been entered, the user may desire that the fate of the changes in a window not be the same as that of the transaction. This is accomplished by leaving the transaction.

5.6. Share Transaction. If two or more windows are open to the same file (or in a multi-user environment, two or more users have a window open to the same file) the user who begins the transaction can designate that the transaction be shared. I.e., the file will appear in all windows as if the transaction were going to be committed.

5.7. Exclusive Transaction. By contrast to a shared transaction, this option provides that other windows open to the file will see it as it was before the transaction began no rows in the file that have already been accessed by the transaction can be changed until the transaction ends.

5.8. Scan Transaction. This function allows the creation of a scan (see below) of all rows that have been changed since the transaction began. (If a row has been deleted, it will not be included in the scan even though it will be un-deleted if the transaction is aborted.)

6. Relations.

Relations (tables) are the common format in which all files are viewed. A relation consists of zero or more columns, each having a unique name within the file (Documents, i.e., tables with zero columns, do not have column names and cannot be the object of functions that require column names).

6.1. Create Relation. New relations can be CREATEd. To do so, the user responds to the prompt "Header:" by typing dashes (-) followed by the column name followed by more dashes and a vertical bar (¦) as the column separator. The dashes are optional as in the following example:

```
----Name----¦--------Address----------¦AREA¦-ZIP-¦
```

6.2. Align Column(s). The data in the columns can be re-ALIGNed - either left, right or centered.

6.3. Sort by Column(s). The rows can be SORTed according to the value of one or more columns.

6.4. Add Column(s). A new column can be ADDed TO a relation. The system will prompt for the name of the column after which the new column(s) is to be added and then prompt for the header which is specified in the same way as for creating a relation.

6.5. Drop Column(s). An existing column can be DROPped FROM a relation.

6.6. Change Column. An existing column can be CHANGEd. The system will prompt for a new header as in adding a column. The new header may be smaller or larger in width than the old one.

6.7. Switch Columns. Two existing columns may be positionally interchanged.

6.8. Unique. This command removes adjacent rows from a relation that have identical values for the specified columns. In response to the "Unique:" prompt the user types column1, column2,...columnN. Specifying "s" in response to the "Options:" prompt will cause the table to be sorted on those columns before the command is executed.

7. Queries.

Queries (searches) may be executed by a user upon a file. The results of a query may be represented by: positioning the tables in the windows (the default); creating a scan; or writing to another window. All query types involve matching of expressions to values and the following wild-card syntax is supported: "\?" matches any single character; "\#" matches any single digit; "*" matches any string of characters; "\$" matches any string of digits; and "\\" matches "\". Furthermore, all query types support the following "Options:"

"a" - locate all occurrences (the default is to locate just the first or next occurrence)

"o" - output the results (the system will prompt for the window number)

"s" - include the results in a scan

"x" - execute the search immediately (the default is to wait for an explicit command to begin) "i" - ignore case (the default is case sensitive)

"w" - whole word matches only, i.e., the matching value must be delimited at both ends (e.g., by space, comma, period, etc.).

Once a query has been defined it may be re-executed using either the NEXT entry on the menu or the Ctrl-L key [2].

7.1. Find. FIND locates occurrences of a specified value anywhere in a file. In response to the prompt "Find:" the user types the pattern followed by CR.

7.2. Replace. REPLACE locates occurrences of a specified value anywhere in a file and replaces it with a specified value. In response to the prompt "Find:" the user types the pattern followed by CR. Then, in response to the prompt "Replace with:" the user types the value to be substituted and CR. There is an additional option unique to the Replace command. The option "n" indicates to perform the replacement without confirmation by the user. The default is to ask the user for each occurrence whether to perform the replacement or not.

7.3. Keyword. KEYWORD locates the conjunction of specified values in the same row anywhere in the file. In response to the prompt "Keyword:" the user types value1 & value2 &...& valueN.

7.4. Select. SELECT locates rows whose column values match a specified value. In response to the prompt "Select:" the user types column1 = value1 & column2 = value2 &...& columnN = valueN.

7.5. Project. PROJECT takes all the rows from a table but only the specified columns and writes them to another window. The "a" and "o" options are both default and mandatory. In response to the prompt "Project:" the user types column1, column2,...,columnN.

7.6. Join. JOIN concatenates two tables based on matching values in the specified columns of each. In response to the "Join:" prompt the user types A.column1 = B.column1 & A.column2 = B.column2 & ... where A denotes the number of the window to be joined to B which is the current window.

7.7. Union. UNION combines two tables of the identical format into a single table. The "a" and "o" options are default and cannot be overridden. In response to the prompt

"Union:" the user types the window number containing the tables to be unionized with the table in the current window [3].

7.8. **Difference.** DIFFERENCE searches two tables of identical format for rows that are based on not matching values in the specified columns of each (i.e., a row in either table qualifies if there is no row in the other table that has equal values in the specified columns). In response to the "Difference:" prompt the user types A.column1 # B.column1 & A.column2 # B.column2 & ... where A denotes the number of the window to be compared to B which is the current window [4].

7.9. **Next.** NEXT re-executes the currently defined query.

7.10. **Clear.** CLEAR clears the definition of the currently defined query.

8. Scan.

A scan is a subset of a table. It can be produced by a query, an engine, or the user. A scan is also persistent; i.e., the scan is not lost when a file is closed. Only one scan per table is presently supported. When a scan exists, those rows that are included in the scan are displayed in **highlighted** text on the screen. Scans can be created in any type of file including Documents. In this capacity, it similar to the "block" commands found in word processors but is slightly more flexible in that while a "block" must be contiguous, a scan can include any lines scattered throughout the file.

8.1. Include. A user can manually INCLUDE the current row (the row the cursor is on) in a scan.

8.2. Exclude. A user can manually EXCLUDE the current row from a scan.

8.3. Begin/End. A user can manually include a contiguous set of rows in a scan by Beginning a scan at the current row, moving the cursor to some subsequent row, and ENDing the scan. This will include all the rows, inclusively, between the begin and end commands in the scan. Once a

BEGIN command is issued, an "S" will appear on the Status Line until the END command is issued.

8.4. Clear. A user can CLEAR a scan; i.e., exclude all the rows from the scan.

8.5. Read. A user can READ a scan from another window into the current window. If the scan to be read is from a table, the current window must either be a table of identical format or empty.

8.6. Write. A user can WRITE a scan in the current window to another window. If the current window is a table, the destination window must be a table of identical format or empty.

8.7. Next/Prior. A user can position the cursor at the NEXT or PRIOR row of a scan.

8.8. Delete. A user can DELETE a scan in its entirety from the table in the current window.

8.9. Move. A user can MOVE a scan in the current window to another place in the file. All the rows in the scan, whether contiguous or not, will be deleted from their present position and inserted contiguously at the new position somewhere else in the table.

8.10. Copy. A user can COPY a scan in the current window to another place in the file. All the rows in the scan, whether contiguous or not, will be inserted contiguously at the specified position in the file.

9. JSpaces.

Judgement Spaces are "inferential indexes" through which queries may, at the user's option, be resolved; i.e., the "matching" of expressions is not done by lexical analysis, but by locating values that are close together in a multi--dimensional factor space based on user judgments. A JSpace is created from a table whose columns represent a set of user-selected variables and whose rows correspond to the rating of rows (or a sample of rows) from an existing table against each variable [5].

9.1. Create. A user can create a JSpace index. In response to the prompt "Create Index:" type the name of the table that contains the user judgements. if no name is specified (i.e., typing <CR> in response), the system will create an ordinary lexical index. In response to the prompt "Length:" the user types the size, in characters, of the terms (keys) in the current window to be indexed. If no length is specified (i.e., typing <CR> in response), the system will use the length of the row as the default. Note that, unlike the query commands, there is no reference to column names. The progress of JSpace creation is displayed on the status line.

9.2. Remove. A user can remove a previously created JSpace by typing its name in response to the prompt "Remove Index:". If no name is specified, the default lexical index is removed.

9.3. Open. A user can open a JSpace by typing its name in response to the prompt "Open Index:". If no name is specified the default index will be opened. This means that all matching done by queries and engines will be done via the open JSpace. While a JSpace is open, "IX:jspname" appears on the status line.

9.4. Close. A user can close a JSpace by typing "Y" or "y" in response to the prompt "Close Index: (Y/N)?"

9.5. Sync. A user can cause an index to be synchronized by typing a "Y" or "y" in response to the prompt "Synchronize Index (Y/N)?" This causes any changes to the index that have yet to be made asynchronously to occur immediately. Synchronization is not permitted during a transaction.

10. Engines.

Inference engines are, in effect, very complex queries; complex principally in the fact that they cannot be quantified. I.e., unlike relational queries for which the criteria can be specified in advance, the criterion is dynamic and self-modifying as the query proceeds [6]. The system has only one such engine presently implemented, the CLOSURE engine, but several more are anticipated as a result of the Phase I investigation and will be briefly described as if they existed [7].

10.1. **Closure.** The Closure engine takes one row of single table and produces the closure [8] of that table based upon two or more columns and constant values. While Closure will accept any table as input, it is only sensible when the table denotes a part-whole relationship such as a work breakdown in which one column represents the Task Name and the other represents the Subtask Name and the Subtask also appears somewhere else in the table as another Task with its own Subtasks, ad infinitum. In this case, Closure can either list all of the Subtasks (and their Subtasks, etc.) required for a Task, or for a given Subtask, list all the Tasks (and Tasks of which those are Subtasks, etc.) of which it could be a Subtask. In response to the prompt "Closure:" the user types A.column1 = A.column2 & A.column3

= A.column4 & ... & A.columnK = A.columnJ where A denotes either the number of the current window or, optionally on the right side of the equal sign, a backslash "\". The option "o" is default and cannot be overridden. The result is obtained by writing the current row to the destination window and then, iteratively: (a) joining the current row of the destination window with the current window based on the column matches; (b) appending the rows in the current window that qualify to the table in the destination window; and (c) joining the next row in the destination window as in (a). This process continues until there is no next row in the destination window. Expressions of the type A.columnK = \.valueK are interpreted to mean that, in addition to the column matches specified, column K must have value K. The table in the destination window is appended with a column with the header of --^--! which indicates the level in the part-whole hierarchy at which result occurs.

10.2. **Match.** The Match engine produces a join between a specified table and the lowest (or highest) level subset of a Closure for which the join is non-empty. Using the Task/Subtask relation and a table that relates Tasks and Task Managers, Match produces a table of all the lowest (or highest) level Managers that must approve a change to a specific Subtask.

10.3. **Part-Whole.** The Part-Whole engine performs a Closure and Match for every entry in a part-whole relation at a specified level (or set of levels or all levels). Given the Task/Subtask relation and a table relating some subset of the Tasks and their Status (e.g., completed, in-work, etc.) the Part-Whole engine produces a table of every Task in the project and its status (the status of some Tasks may not be inferable based on limited information).

10.4. **Deviance.** The Deviance engine performs a difference between a specified table and either a Match or a Part-Whole. Given the Task/Subtask relation and the Status table above and a table relating each Task with its scheduled Status, the Deviance engine produces the list of Tasks that are at variance with the schedule.

10.5. **Temporal.** The Temporal engine takes a specified table (or subset of it) and produces a table that represents a "wait-for" graph. While it can be applied to any table, it is only sensible if the table denotes a precedence relationship such as a table relating a Task and its Status to another Task and its Status in which the first [Task, Status] is only possible when the second [Task, Status] is true (Status can be any combination of columns including numerical data). Given such a table, the Temporal engine will produce a table listing each Task, the Tasks it is waiting for, the Tasks those are waiting for, etc.

10.6. **Resource.** The Resource engine takes a specified table (or subset of it) and produces a table that represents a queue. While it can be applied to any table, it is only sensible if the table denotes a consumption relationship such as a table relating a Task and its Status to a quantity of a resource in which the [Task, Status] is only possible when the quantity of the resource is available (Status can be any combination of columns including numerical data). Given such a table, the Resource engine will produce a table listing each Resource, the Tasks waiting for that resource, the Tasks waiting for those Tasks because they need that resource too, etc.

10.7. **Begin.** Begin prompts the user to select an engine, specify the parameters for that engine, and starts the engine processing.

10.8. **End.** End stops an engine and clears its parameters.

10.9. **Checkpoint.** Checkpoints suspends an engine and causes it to write a record of all parameters it requires to resume processing from where it left off.

10.10. **Restart.** Restart prompts the user to select an engine which was checkpointed, and causes the engine to read its restart record and resume processing.

FIGURE 1 - TYPICAL SCREEN

```
|1|KSC\DATA\INSTAL,255,124     A I S  ** TX Log:0     Cap:1394 **

 Scan  Query  JSpace  Transact  Window  File  Relation  Engine

|1 BE-NUM      CAT         INSTALLATION NAME                        CC   ADD
0520-00017   30111 |DURNBURG RR AND HIWAY RIVER BRIDGE          |EG |400
0520-00086   36130 |HOHEN DAM                                   |EG |400
0520-00086   30130 |HOHEN HIWAY BRIDGE                          |EG |400
0520-00263   90404 |BAD LANGENGALIZA MUNITION STOR DEPOT        |EG |400
|2 BE-NUM                        DESCRIPTION
0520-00017   MAJOR RIVER CROSSING CONSISTS OF TWIN RR AND HIWAY
             BRIDGES WITH HIWAY BRIDGE OVER RR BRIDGE.  HIWAY
             BRIDGE IS FOURLANE.  RR BRIDGE IS TWO TRACK.  THREE
             SPAN CONCRETE ABUTTED CONSTRUCTION WITH LARGE CON-
             CRETE PILINGS IN RIVER.  FOUR (4) ZSU-57-2 REVETTED
             EMPLACEMENTS AT WEST END OF BRIDGE AND SIX (6) ZSU-
             57-2 REVETTMENTS AT EAST END.
0520-00086   HYDROELECTRIC DAM, STEEL REINFORCED CONCRETE CON-
             STRUCTION WITH TWO TURBINE SLUICEWAYS IN CENTER OF
             DAM.  STEEL-GATED SPILLWAY AT EAST END.  FOUR LANE
             HIWAY ON CREST OF DAM.  EIGHT (8) ZSU-23-4 HARDENED
             REVETMENTS AT EAST END AND TWELVE (12) ZSU-57-2
             HARDENED REVETMENTS AT WEST END.
0520-00086   FOUR LANE CONCRETE ROAD ON CREST OF HOHEN DAM WITH
              WIDE TURN-OUTS AT EITHER END.AREA IS HEAVILY DEF-
```

FIGURE 2 - PULLDOWN MENUS

```
|1|TEMP1,127,1          IX:JS1 A I S  ** TX Log:0     Cap:1394 **

 Scan  Query       JSpace  Transact   Window  File  Relation  Engine

 Incl  Find        Create  Begin      Selec   Ope   Crea      Closure
 eXcl  Repla       Remove  Commit     Open    Cloe  aLig      Match
 Begi  Keywo       Open    Abort      Close   Rea   Sort      Part-Whole
 End   Selec       Close   Enter      cleaR   Wrie  Add       Deviance
 cLea  Proje       Sync    Leave      Link    Top   Drop      Precedence
 Read  Join                Share              Boto  cHan      Consumption
 Next  Union               eXclusive          Syn   sWit      Begin
 Prio  Difference          Scan                     Uniq      End
 Dele  Closure                                                checKpoint
 Move  Next                                                   reStart
 Copy  Clear
 Writ
```

11. Miscellaneous Commands.

All of the commands that can be executed via the pulldown menus may also be executed by a one or two key control sequence. There are many additional commands that can only be executed by a control key sequence. The following is the complete command set currently implemented.

11.1. One Key Commands.

These are the commands accessed by holding the CTRL key down while typing a single character or pressing one of the specially designated keys (e.g., TAB) on the keyboard.

^B or Shift-Tab: Tab to the previous column of a table or a previous position in a document. If the system is in Auto-Tab mode (an "A" appears on the Status Line), the cursor will position under the first non-blank character in the previous row.

^C or PgDn: Page Down; i.e., move down the file one window length.

^D or Rt Arrow: Cursor Right; i.e., move the cursor right one character.

^E or Up Arrow: Cursor Up; i.e., move the cursor up the file one row.

^G or Del: Delete Right Character; i.e., delete the character under the cursor and shift all the remaining characters after the cursor left one position.

^H or BS: Delete Left Character; i.e., delete the character to the left of the cursor and shift all the remaining characters after the cursor left one position.

^I or Tab: Tab to the next column of a table or the next position in a document. If the system is in Auto-Tab mode (an "A" appears on the Status Line), the cursor will position under the first non-blank character in the previous row.

^J or Home/End: Beginning/End of Row.

^K: Prefix to the set of two key commands that begin with ^K. It is not necessary to hold the CTRL key down for the second character.

^L: Repeat Last Query; i.e., execute the currently defined query in the current window again. See the endnote on query processing for a more complete description of how the ^L command operates.

^M or CR: New Row; i.e., insert a new blank row between the row the cursor is on and the following row. Characters to the right of the cursor, if any, will be deleted from the current row and moved to the new row automatically.

^N: Insert Row; i.e., similar to CR except that no characters will be deleted from the current row and the new row will be empty.

^O: Prefix to the set of two key commands that begin with ^O. It is not necessary to hold the CTRL key down for the second character.

^P: Insert Control Character; i.e., in order to place a control character in the data (e.g., a printer command), type Ctrl-P and the next character will be interpreted as a control character. This is <u>not</u> necessary for macros. The macro command processor interprets any upper case letter as a control character and anything else as itself.

^Q :	Prefix to the set of two key commands that begin with ^Q. It is not necessary to hold the CTRL key down for the second character.
^R or PgUp:	Window Up; i.e., move backward in the file one window length.
^S or Left Arrow:	Cursor Left; i.e., move the cursor to the left one character position.
^U:	Abort Any Command in Progress. While typing on the Command Line, the system will act as if the command had never been started. If the command (e.g., a search) is already in progress, the system will stop processing at the first occasion in which everything is properly synchronized.

^V or Ins:	Toggle Inset/Typeover Mode; i.e., in Typeover Mode the character typed will replace the character under the cursor; in Insert mode (an "I" appears on the Status Line) the character typed will be inserted under the cursor and all characters to the right of the cursor will be shifted right one character position.
^W:	Scroll Window Up; i.e., move the window backward one row in the file.
^X or Down Arrow:	Cursor Down; i.e., move the cursor to the next row in the file.
^Y:	Delete Row; i.e., remove the row the cursor is on from the file.
^Z:	Scroll Window Down; i.e., move the window forward one row in the file.

11.2. ^K Prefix Commands. These are the commands accessed <u>after</u> typing Ctrl-K. It is not necessary to hold the CTRL key down when typing the second letter of the command. It will always be interpreted as a control character.

^KB: Begin Scan (previously described).

^KC: Copy Scan (previously described).

^KE: Exclude Row from Scan (previously described).

^KG : Read Scan (previously described).

^KH: Clear Scan (previously described).

^KI: Include Row in Scan (previously described).

^KK: End Scan (previously described).

^KL: Prior Row in Scan (previously described).

^KN: Next Row in Scan (previously described).

^KO: Open File in Window (previously described).

^KP: Write Scan to Window (previously described).

^KR: Read ASCII Text File (previously described).

^KT: Define Tab Width; i.e., set the number of character positions to tab in a document when not in Auto-Tab mode.

^KU: Abort Command (same as ^U).

^KV: Move Scan (previously described).

^KW: Write ASCII Text File (previously described).

^KX: Quit and Exit System (previously described).

^KY: Delete Scan (previously described).

^KZ: Close File (previously described).

11.3. ^O Prefix Commands. These are the commands accessed after typing Ctrl-O. It is not necessary to hold the CTRL key down when typing the second letter of the command. It will always be interpreted as a control character.

^OA:	Abort Transaction (previously described).
^OB:	Begin Transaction (previously described).
^OC:	Commit Transaction (previously described).
^OD:	Leave Transaction (previously described).
^OE:	Enter Transaction (previously described).
^OG:	Select Window (previously described).
^OI:	Tab to Next Column (previously described).
^OJ:	Link Window (previously described).
^OK:	Change Case; i.e., if the character under the cursor is lower case it will be changed to upper case and conversely.
^OL:	Center Text; i.e., center the text under the cursor in the column.

^ON:	Include Transaction in Scan (previously described).
^OO:	Open Window (previously described).
^OS:	Share Transaction (previously described).
^OU:	Abort Command (same as ^U).
^OW:	Select Window Up; i.e., make the window above the current window the new current window.
^OX:	Exclusive Transaction (previously described).
^OY:	Close Window (previously described).
^OZ:	Select Window Down; i.e., make the window below the current window the new current window.
^O1...^O9:	Select Window 1...9 (previously described).

11.4. ^Q Prefix Commands. These are the commands accessed after typing Ctrl-Q. It is not necessary to hold the CTRL key down when typing the second letter of the command. It will always be interpreted as a control character.

^QA:	Find and Replace (previously described).
^QC:	Position at Bottom of File (previously described).
^QD or End:	Position at End of Row (previously described).
^QE:	Clear Window (previously described).
^QF:	Find String (previously described).
^QG:	Add Column (previously described).
^QH:	Change Column (previously described).
^QI:	Toggle Auto-Tab Mode; i.e., if Auto-Tab Mode is on (an "A" is on the Status Line) then turn it off. If it's off, then turn it on.

^QJ:	Align Column (previously described).
^QK:	Drop Column (previously described).
^QL:	Keyword Search (previously described).
^QN:	Select (previously described).
^QO:	Join (previously described).
^QP:	Project (previously described).
^QQ:	Clear Query (previously described).
^QR:	Position at Top of File (previously described).
^QS:	Position at Beginning of Row (previously described).
^QT:	Switch Columns (previously described).
^QU:	Abort Command (same as ^U).
^QV:	Sort Relation (previously described).

^QW: Unique (previously described).

^QY: Delete to End of Row; i.e., delete all data to the right of the cursor.

^QZ: Closure (previously described).

12. Macro Commands.

There is a very primitive macro processor which is activated by F9. It does not allow any parameter substitution but merely processes the contents of the file as a series of keystrokes. To create a macro, simply type the sequence of keystrokes as an ASCII text file using capital letters to represent control characters. This can be done from within the system by using the Write File command to create an ASCII file.

13. Installation.

13.1. The system requires a PC or 100% compatible with MSDOS 3.x or higher. Most of the popular video boards (Mono, CGA, Hercules, Paradise, EGA) are supported but the system does direct video output so it expects screen memory to be at $B800 for color or $B000 for monochrome.

13.2. The system will operate with as little as 256K but some functions will fail due to a lack of dynamic memory. It is fully functional at 384K but will make use all available memory up to 640K for buffer caching.

13.3. The system doesn't care much about directory structure and it supports path names including ..\ notation. However, the files TT.EXE and EDITERR.MSG must be in the same directory and that directory must be included in the MSDOS PATH specification. The current working directory is the default directory for all files unless a path name is specified. User file names may not have extensions as the system assigns its own extensions (.DEF, .DES, .TXT, and .IX#).

14. References.

[1]. Ossorio, P.G., Schneider, L.S., Final Technical Report, Contract F-30602-85-C-0190, Rome Air Development Center, Griffiss AFB, NY, 1987.

[2]. Queries execute off a stack which is particularly important to know when executing joins. A join is specified by an expression such as 1.TASK = 2.TASK. This defines a join on window #2 such that the top row in window #2 will have its TASK equal to whatever the TASK is on the current row (the row the cursor is on) in Window #1. However, the synchronization is not continual but only occurs when a ^L is executed in Window #1. For example, if a Select is defined in Window #1, then each time a ^L is executed, the next row qualified by the Select Expression is found, and the Join from Window #2 to Window #1 is re-executed. But it is not necessary that any query be defined in Window #1. A ^L will always cause the stack to execute. I.e., as long as the join is defined, you can simply position the cursor in Window #1 and issue a ^L command and Window #2 will be repositioned. Furthermore, the stack can be of any practical depth such as joining #2 to #1, #3 to #2 and #4 to #3 and #5 to #3. This creates a tree of joins with #1 as the root. Henceforth, any- time a ^L is issued in Window #1, #2 will be repositioned. Then, because #3 is joined to #2, it will be repositioned, etc., until all the joins in the stack have been executed (except circular joins, which will only execute once).

[3]. As of this writing, the UNION command does not work properly and is still undergoing tests. However, a Union can be accomplished by creating a Scan of all the rows in one table and then reading that scan into the other table. See the paragraph on Scans below.

[4]. As of this writing the Difference command is not working properly and is still undergoing tests. However, it can be accomplished by performing a Join using the "s" (scan) option, and then deleting the scan and writing what is left to another window. I.e., only the rows that did not join will remain which is the desired result (this must be done under transaction management or the original table will be corrupted.)

[5]. The knowledge structure that comprises JSpaces is that of a Factor Space. In contrast to lexically-based technologies, factor spaces provide a psychometrically based indexing method that bypasses the usual limitations of word-shape or mutually exclusive indexing categories. This is achieved by the construction of a multi-dimensional space in which each dimension represents a significant variable that discriminates among, for example, resources in the project (skills, computer systems, dollars, individual persons, etc.); and each is represented by a vector of coordinates that locate it, geometrically, in the factor space. This allows, for example, the resources necessary for a process to be retrieved according to the principle that resources located closer to each other in the factor space are more similar to each other in terms of the factors comprising the space.

In the problem domain of scheduling, factor spaces might be constructed to represent, at least: (a) attributional similarity among resources as above; (b) transformational similarity among paradigms; (c) part-whole similarity among processes and achievements; and (d) achievement similarity between processes and process similarity between achievements. In all cases, the general procedure that a user would employ to construct a factor space is as follows: (1) Select a set of variables that jointly discriminate within the problem domain the representation units or elements thereof; (2) Select a sample of the representation units among which discriminations will have to be made (e.g., vocabulary terms, paradigm case descriptions, constituent and/or process definitions); (3) Create a table (matrix) of the sample with respect to the variables and record judgments about the relevance of each sample item to each variable (this can be qualitative such as "highly relevant" or quantitative on a numeric scale); (4) Invoke a statistical process provided within the system to analyze the correlations among the variables (this eliminates commonalties among the variables and creates a set of orthogonal axes that form a space with the required geometric properties); (5) Have the system scan all of the representation units (or elements thereof) in the domain of interest and assign to each a vector of coordinates representing its location in the factor space; (6) Henceforth, when desired, request the system to assign coordinates to a hypothetical representation unit or element thereof (i.e., a request for one that is needed but may or may not be present) and to retrieve whatever units do exist in the order of their distance from the coordinates of the request.

The normal operational cycle of a factor space is then roughly as follows: new representation units or elements thereof are created by the user; these are scanned by the system and assigned a coordinate vector in the factor space based upon how it relates to others already located in the space; the user or the part-whole inference engine interrogates the factor space for a list of representation units or elements that are relevant to a needed representation unit or element; and the factor space is monitored by the system for unrelated or indistinguishable representation units so that the analyst can periodically add or remove variables.

Factor space indexing provides several unique features. Indexing would be accomplished automatically, relieving the user of the tedium of systematically coding the representation units and elements. Also, representation units and elements would be retrieved by the system on the basis of the user's judgments about content, freeing him from adhering solely to searches based on the lexically structured query and inference engine capabilities already proposed. Finally, representation units and elements would be, at the user's option, retrieved in order of relevance (as opposed to simply "qualified" or "unqualified") providing a formal basis for estimating the degree of relevance between the planned schedule and what is actually occurring, hence, the degree of confidence that a project is or isn't proceeding according to plan (and the reasons for that conclusion).

[6]. Formally such "queries" are considered to be beyond the power of first-order logic in that the quantification of the result after i iterations is dependent on the result of iteration (i - 1). In this sense, they are motivated (fueled) by the results produced along the way (and, perhaps, this is one justification for referring to them as "engines").

[7]. The technical requirements of these are described in the other sections of this report. They are redescribed here from a user's perspective. In the technical discussions, however, they are referred to as extensions to a single inference engine. Experience with the Phase I prototype, however, suggests that from a user's standpoint, a modular implementation of several engines is advantageous in two regards: (a) it retains the closed nature of the system in that every operation produces an object (table, scan, etc.) that can be manipulated by all the other functions of the system; and (b) a user has the flexibility to combine the engines in a number of ways, some of which may not be obvious even to the system designers.

[8]. Given a relation R(...,A,...,B,...) having two attributes over a common domain, it is then possible to have joins of indefinite length: R [A=B] R [A=B] R [A=B]... and in the general case the topological structure defined on the tubles of R is a digraph. We can invent a notation within relational algebra such that R^n[A=B] means R [A=B] R ... [A=B] R where R occurs n times. But there is no way to allow this to occur an indefinite number of times within relational algebra. We define the Closure operator R^+[K,A,B] as follows where K is a key of R, and A and B are attributes or lists of attributes having the same domain:

> R^+[K,A,B] is the set of all tuples $\langle K_1,K_2,n\rangle$ where there is a join sequence on A=B of length exactly n.

The Closure operator combines with other operators of relational algebra in such a way that it may occur on any semantic loop definable within the schema (e.g., upon a projection of a compound expression formed of joins). It has great import for part-whole relations such as determining the lowest common superior of a set of elements of a hierarchy, or the highest discriminants. For example, in the following hierarchy, the commonality between Algeria and Uganda is Africa, and the highest discriminants are (North Africa, East Africa).

```
1. Africa
     1.1.       North Africa
                1.1.0.1.    Mediterranean
                     1.1.0.1.1.      Algeria
                     1.1.0.1.2.      Libya
                1.1.0.2.   Atlantic
                1.1.0.3.   Interior
     1.2.       West Africa
     1.3.       South Africa
     1.4.       East Africa
                1.4.0.1.   Kenya
                1.4.0.2.   Uganda
                1.4.0.3.   Tanzania
```

Where the structure is a lattice rather than just a hierarchy, closure can determine the common inferior and its discriminants.

A Knowledge Dictionary System for Scheduling Support

P.G. Ossorio and L.S. Schneider

Appendix B

Submitted by
Linguistic Research Institute, Inc.
5600 Arapahoe Avenue
Boulder, Colorado 80303

Submitted to
Rome Air Development Center
Griffiss AFB, New York

TABLE OF CONTENTS

1. OS/LAN Compatibility.

Operating System (OS) software and Local Area Net (LAN) software must be compatible for an implementation to work correctly. A parametric analysis of each of these and their interface topology is developed and explained; and a possible compatibility matrix is suggested.

2. Operating System Software.

The Operating System (OS) software traditionally provides the interface between the system hardware and the most immediate user of that hardware; typically a programmer. The possibility of "end users" in the traditional sense being users of the OS is not considered in this analysis [1] and the term "user" consistently denotes a program or programmer throughout this discussion. The issue of whether the LAN software is an OS user is considered in another section. The principle conceptual entities dealt with by an OS are processes, memory, context, files and inter-process communication (IPC). Its purpose is to issue instructions to the Central Processing Unit (CPU) to implement and manage these entities.

2.1. Process. To the OS, a process consists of a sequence of instructions (code) and data over which the instructions operate. Such a collection is typically stored as a file and does not acquire the status of a process until the OS has received a request from another process to execute that file. The problem of an infinite regression of processes is solved by what is known as a "bootstrap" program; i.e., a program whose name is encoded within the OS as the first process to commence after the OS is loaded and initialized [2]. The issues related to processes are initiation, execution, and location of both the code and data segments.

2.1.1. Initiate. The ways in which processes can be initiated by the OS are legitimate parameters for describing its behavior. An OS must, of course, have at least some way to initiate a process; and most have several. The principal variants are as follows.

2.1.1.1. Spawned. A process can be "spawned" by another process; i.e., can be converted from the file in which it resides into a process waiting for execution. Every OS must have at least this capability.

2.1.1.2. Forked. A somewhat more advanced way of initiating a process is by a "fork." In this case, a process that is already running recreates itself as another process waiting for execution. This is most common technique in multi-user systems, particularly for the shell or command line interpreter. Every time a new user logs into the system, the shell reinstantiates itself as another process to serve that user and continues executing until that user logs out.

2.1.1.3. Primed. In performance intensive systems with many users logging in and out at very high frequencies (e.g., a system serving Automated Teller Machines) a technique called "priming" is employed. In this case, due to the overhead involved in forking, a process forks repeatedly at initialization, creating a specified number of "primed" processes that wait for users and do not terminate when the user logs out, but merely reinitialize and wait for another user. Primed processes do not terminate until their termination is requested explicitly by another process.

2.1.2. Execute. The way in which processes are executed after initiation is also characterizes the behavior of the OS. It is possible, but unusual, for an OS to have more than one way of controlling process execution.

2.1.2.1. Dedicated. "Dedicated" execution is the environment in which whatever process is executing has the attention of the OS for as long as it wishes, exclusive of any other processes. In this environment, the only disruption of a process is a hardware interrupt or trap that causes the program counter to be reset to a value stored in a predefined memory location associated with that hardware interrupt line. Such vectors are almost always to code in the OS (e.g., to service a disk drive during data transfer) although nothing prevents a process from changing these vectors. Dedicated execution is predominant among extremely simple, single-user systems, few of which remain in existence today.

2.1.2.2. **Sliced.** "Sliced" or time-slicing was the first and simplest approach to multi-user servicing. In this approach, the OS associates a rundown timer with each process in execution and sets that timer to an equal or apportioned "slice" of time (e.g., 100ms). Then, every time the OS gets control via a hardware interrupt, it updates the timer based on a real-time clock and, if it finds that the value of the timer is 0 or negative, instead of returning control to the interrupted process, it returns control to the next process and resets the timer for the interrupted process. Sliced execution is only found today in either very dated systems (due to its simplicity), or in very modern super-computers (due to its speed).

2.1.2.3. **Priority Interrupt** . By far the most common form of execution control, the "priority interrupt" approach utilizes special hardware interrupts found in almost all modern CPU's to accomplish time slicing. Each process is assigned to a frequency interrupt register according to its priority over other processes. The hardware continually generates interrupts for each register and thus both enables the OS to get control at specified intervals and is able to stack interrupts according to priority when an interrupt is generated during the processing of another interrupt. This allows the OS to take maximum advantage of "idle" time (time in which a process is waiting, e.g., for a disk transfer) in allocating it to processes that are ready to resume execution.

2.1.2.4. Stacked. "Stacked" execution is found today only in extremely high-end mainframes, although the technology is by no means new. The stacked approach achieves advantage over the interrupt approach in much the same way that primed processes do over forked processes. The OS determines ahead of time which processes are to execute and how frequently (including itself) and pushes all the information onto a hardware stack. Then, every time an interrupt is generated, the hardware itself pops the information necessary to activate the process off the stack and relinquishes control to that process. In this approach, the OS is not a privileged process, but has the same status as all other processes. Its ability to control the system derives only from the fact that it is the first process to execute and thus can initialize the stack to prioritize its own execution.

2.1.3. Code Segment. The way in which the OS manages the code segment of a process is a characteristic of importance, particularly with respect to performance. Most systems support some admixture of the following.

2.1.3.1. Duplicate. Duplicating the code segment for every instantiation of the process is the oldest, simplest, but still the fastest approach. The code does not have to be re-entrant [3] and there is little to do when switching from one process to another (see Context below). It is expensive in terms of memory, which was a prime motivator for developing the other approaches, but is resurgent today as memory costs have fallen through the floor.

2.1.3.2. Single re-entrant. In this approach, there is only one copy of the code segment in memory, regardless of how many processes are instantiated from it. Ano, not surprisingly, this technique predominates in contemporary systems with limited memory. The price paid is, of course, that an enormous amount of information must be saved when the process is interrupted, and the same information retrieved when the process restarts, all of which takes time and is referred to as "context switching" (see below).

2.1.3.3. Duplicate re-entrant. This environment is found, again, mostly in high-end mainframes. It is an attempt to gain the advantages of both duplicate and re-entrant execution in a way that makes optimum use of the system. The OS monitors how frequently each code segment is being referenced by different processes, and optionally generates duplicate code segments as needed or collapses duplicate code segments into a single segment as demand decreases. Generating new duplicates is very easy and very fast, which is ideal in that the need to do so occurs when the workload is increasing very rapidly. Collapsing duplicates is difficult and slow, but that is of little consequence since the need to do so is prompted by the fact that the system workload has become very relaxed.

2.1.4. Data Segment. Managing the data segment of a process is also characteristic of an OS but is also highly correlated with the way in which the code segment is managed.

2.1.4.1. Process-bound. In this approach, all data is bound to the process, and only indirectly referenced by the code segment. This approach is required if code segments are allowed to be completely re-entrant. The principle advantage is in the insulation among users that is obtained. The code segment knows nothing about multiple users and each process for which it executes has the illusion of being the only process that exists. The elegance is paid for in terms of the high degree of indirection (with its attendant performance cost) necessary to access data.

2.1.4.2. Code-bound. In this approach, some or all of the data may be allowed to be directly bound to the code segment at the user's option (or as may be determined by an optimizing compiler). This provides for having only one copy of constant and global data, while maintaining seperate copies of only those variables that are process--dependent. A primary exemplar of a process requiring code-bound data is a database management system. It is absurd to think of duplicating (and, hence, maintaining consistency among) all the data dictionary and data definition variables when only a handful of variables (e.g., the query) are specific to the process. In fact, in systems that do not allow code-bound data (e.g., Unix), implementers of database systems have without exception utilized files or raw I/O (see below) to bypass the OS.

2.2. Memory. Memory is the major asset of the hardware that must be managed by the OS. Memory, as used here, denotes only that store directly addressable by the CPU in the context of executing a CPU instruction. It does not include any extended stores such as RAM under control of a Memory Management Unit (MMU) unless the MMU is invisible to the OS (which is not often). Any data residing in other than memory can be thought of as being on a disk or equivalent storage mechanism that requires extensive effort (and time) before it is accessible to the CPU.

2.2.1. Real. The real memory is the RAM that is electronically connected to the CPU memory bus and can be directly accessed by specifying an address to the CPU. As far as the OS is concerned, it is the fastest and only such memory [4]. The principal characteristic of an OS regarding the management of real memory is how the OS presents that memory to the user. To some extent, the presentation reflects the actual hardware organization, but this is purely a convenience to the OS (and a burden to the user) and is never a logical requirement. Any OS is always, in principle, capable of mapping the hardware memory organization to whatever presentation it desires; its just easier not to.

2.2.1.1. Linear. The simplest presentation of memory to the user is that of a linear sequence of bytes, starting at 0 and ending at the highest number of bytes in the system. When the hardware actually uses such a structure (e.g., the Motorola 68000 series CPU) the OS will almost always present it that way to the user, even if the OS was ported from another CPU that did not use a linear address space. When the underlying structure is other than linear and the OS presents it as linear, the OS will almost always impose a limit on the address space available to the user that, in some way, reflects the non-linear structure. That's why we always have machines with megabytes of RAM on which the largest program cannot exceed 64k.

2.2.1.2. Segmented. A segmented presentation is always the most difficult for the user and, if the hardware memory is segmented, the easiest for the OS. Segmented memory is a linear memory that does not allow linear addressing, but instead requires a structured address such as <bank, page, paragraph> in which each of the components is modulo. The user has no choice as to where his memory begins and ends, but it is the users responsibility to check for page boundaries. E.g., on an 80X86 CPU, the address of the byte that follows 00FF:0FFF:F is 0100:0000:00.

2.2.1.3. Paged. A paged presentation is a segmented presentation except that the user always sees pages as if they belonged to him. I.e., he will always own (as he sees it) byte 0 of every page he has, even if that begins in the middle of a hardware page. He thus requests memory in terms of pages and allocates his data in accordance with the page structure he requested and the OS takes care of mapping those addresses to the hardware addresses.

2.2.1.4. **Protected.** Memory protection is a hardware feature that applies to any memory presentation and augments that presentation to serve not only as the mode of addressing, but also as a boundary outside of which any access attempt will trigger a specially designated hardware interrupt if the system is operating in what is called "user mode." What happens as a result of the interrupt is up to the OS, but almost always is a process termination. No process running in user mode can change the mode register because it is outside the users memory. But when the system powers up, it is in "system mode" by default so the OS process begins in system mode and has access to all memory including the mode register. Thus the last instruction the OS executes before activating another process is to put the system in user-mode and all hardware interrupts automatically return in system-mode.

2.2.2. **Virtual.** Virtual memory refers to memory that **appears** to the user as if it were real memory; i.e., he can treat it in the same way as real memory and, in fact, does not even need to distinguish between the two unless he is concerned about performance. The way in which the OS implements virtual memory is one of the most critical characteristics of its performance and its integrity. By definition, virtual memory is much larger than real memory, and the principle problem that confronts the OS is how to map the virtual space to the real space and how and when to act on that mapping.

2.2.2.1. Swap. The oldest and simplest approach to virtual memory is called "swapping." In this approach, the OS generates a load map for a process when it is instantiated; i.e., the set of real memory locations that the process will use **whenever** it is active. The map is stable and does not change from one activation to another. Henceforth, when a process is activated whose load map intersects the load map of another (or several) inactive process, **all** of the memory associated with the inactive process is written to disk and the location of that information on the disk is noted in the load map for that process. Then, the load map for the about to be activated process is interrogated and its disk-based memory image is read into memory according to the load map. To minimize complexity, most systems that employ swapping also set a fixed size limitation on processes so that only one process will have to be swapped for each process that is swapped in. This is known as Multiple Fixed Tasks (MFT) management.

2.2.2.2. Demand Paging. Unlike swapping, demand paging does not swap on a process basis, but on a memory page basis only when necessary. Nothing at all is done when a process is activated. It is only when an active process actually makes a request to access a part of its address space that is not in real memory (called a page fault) that an action is taken. The action is to swap out the hardware memory page regardless of its mapping to processes, and replace it with the image of that page as the process last saw it. Moreover, as there is no reference to processes, any hardware page can be selected for swapping and the memory map for the process will be altered accordingly (i.e., multiple references to the same virtual address may reference different real addresses). Demand paging is typically done according to some strategy (e.g., least recently used or least frequently used) that dynamically avoids "thrashing;" e.g., two pages that reference each other competing for the same real page.

2.2.2.3. Explicit Paging. Explicit Paging is usually provided either as an extension to demand paging, or when no automatic swapping of any kind is provided. An exemplar of the latter case is the use of overlays; i.e., segments of code structured by the programmer in a non-conflicting way and explicitly declared by the programmer to reside in the same real memory segment. In the former case, explicit paging is offered to the user for one of two purposes: (a) to lock a process into memory so that it will not get swapped out, even if it is not very active (e.g., a primed process that is waiting for a user); or (b) to force all pages of a process to be swapped out if any one of the pages are swapped out (e.g., if the user knows a priori that he has two very large processes, neither of which can do anything unless all its pages are resident).

2.3. Context. Given that an OS provides for re-entrant code and process-bound data, the combination of the singular code segment and the data segment associated with one of many processes is called a "context" and a context has to be created, maintained and terminated.

2.3.1. Create. Context creation refers to building what amounts to a process description, including place holders for all the information necessary to record the state of the process when the process is interrupted. A process description will typically include references to: the code segment, the data segment, the stack, the stack pointer, all opened files and their interface blocks, and both the real and virtual memory maps.

2.3.2. Switch. Context switching refers to a four stage process that is probably the single most expensive function (in terms of the percentage of total available CPU time consumed) performed by an OS. First, the entire state of the current context is recorded in the process description and the description and any (or all if the system uses swap management) of the processes memory pages so indicated are synchronized to stable store. Next, the OS restores its own context from its last synchronization point including swapping in any indicated memory pages and commences its own processing. Thirdly, its entire state is recorded and the description and any of its memory pages so indicated are synchronized to stable store. Finally, the next user context is restored from its last synchronization point and it's execution is commenced [5].

2.3.3. **Exit.** The termination of a process is considerably more complex than merely erasing the evidence of its existence. To the extent that the process has acquired resources (see below), those resources must be accounted for, released, and made available to other waiting processes, not the least of which is the OS itself. It is upon process termination that the OS performs most of its scheduling and dequeuing functions.

2.4. **Files.** In modern OS terminology, a file refers to any data stream stored on, sent to or received from anything other than real memory; i.e., files include all the data within the system and in other systems to which the system has access. They are the major asset of the software that the OS must manage.

2.4.1. Data. Data files represent only those types of files that were traditionally called files. These are structured user data (e.g., the files of a database), text (e.g., program source code), and other data that is maintained by user programs. The three primary characteristics that distinguish among OS with respect to files are format, access method and versioning.

2.4.1.1. Formats. The file format refers to the format in which the OS presents the file to the user. It is most often the case that this is also the way in which the OS formats the file on the hardware. However, this is not a requirement and many OS perform some mapping on behalf of the user to hide many of the complexities of internal structure. In the latter case, it is extremely important for the user to understand these complexities at least to the extent that they impact performance. Most OS offer more than one format, allowing declaration by the user at file creation.

2.4.1.1.1. Stream. A stream format is one in which the bytes of the file are sequential and are accessible in order beginning with byte 0. The internal representation of a stream is either very similar (except that it deals with the complexities of segmentation) or is "linked." In the latter case, the bytes within a page are sequential but the pages are asequential and the ordering is maintained as a sequential list of pages [6].

2.4.1.1.2. Text. A text format is a stream in which are embedded certain control characters that enable sequential access in ways other than on a byte by byte basis. The most common is the <CR/LF> (carriage return, line feed) pair. When this is present, the file is accessible one "line" at a time where line denotes all the bytes up to and including the next <CR/LF>. It is possible to embed additional control sequences although this is most often left to the user (e.g., a word processing system will insert control characters for paragraphs, pages, headings, etc.).

2.4.1.1.3. Paged. A paged format is one in which the unit of addressability is larger than a byte, and is so not merely by the presence of control characters as in a text format, but by the imposition of structure. The implementation of a page format requires that a page "size" either defined or default be declared for the file when it is created. It is then accessible on a page by page basis.

2.4.1.1.4. **Structured.** Structured files are identical to paged files at the top-most level, but allow the subdivision of pages into complex components called "fields" which may consist of other fields. The structure must be declared when the file is created. It is then accessible on a page.field.field. .. .field basis.

2.4.1.2. **Access Methods.** The access method refers to the means by which components of a file can be accessed. The access method is obviously not independent of the format inasmuch as the format determines what components exist. It is independent of the format in terms of how components are accessed and modified.

2.4.1.2.1. **Sequential.** The simplest access method is sequential. I.e., component N only becomes accessible immediately after access to component N-1 has occurred, and the first access will always be to the first component if the file is opened for read or read/write, or after the last component if the file is opened for append. Most of the simpler OS access streams and text files in this manner, maintaining a pointer to the location in the file at which the next access will occur, and setting flags for the user when certain conditions are true (e.g., End of File, End of Line, etc.).

2.4.1.2.2. **Direct.** Direct access allows any component of a file to be accessed directly by the invocation of a "seek" function. The parameters to the seek function include the component's position, relative to the first component. The effect of the seek function is to set the file pointer to the end of the preceding component so that the next sequential fetch will obtain the desired component. Internally, the seek function may be implemented as a loop of sequential accesses or the OS may implement a structure on top of the file (e.g., a page map) to support direct reads at the expense of more costly adds and deletes (because the page map requires updating whenever a component is added or deleted).

2.4.1.2.3. **Indexed.** Indexed access augments direct access to support component addressing by value. I.e., the user does not have to specify to relative position of the component, but merely its contents (or a range of contents). Most OS implement indexed access based on the definition of a "key" component; i.e., within the page, a field (if the format is structured) or a range of byte positions and it is this component and only this component that may be used for addressing. As with direct access, a structure is typically implemented on top of the file (e.g., a B-tree) to increase the performance of retrievals and the expense of updates.

2.4.1.2.4. **Indexed Sequential.** This access method is optional depending on how the OS implements indexed access. It is frequently the case that a file accessed directly or by index will also need to be accessed by the user sequentially at times (e.g., an accounting application will need sequential access for end-of-month processing). If the OS implements indexing as a B-tree or a hash table, then no straight forward means of sequential access is possible other than the unacceptably high-cost method of attempting to access every possible key value in ASCII order (99% of the attempts will fail, and the cost of a failure is usually higher than the cost of a successful search). In such cases, the OS may provide an indexed sequential access method that supports both direct and sequential access at an acceptable cost. The form of implementation varies but by far the most frequent in contemporary systems is the B+ Tree (a B-tree in which the leaf nodes are linked from left to right). The cost of sequential access is still considerably higher with a B+ Tree than with a sequential file but are, at least, acceptable for reasonably small files (see Updates below).

2.4.1.2.5. Virtual Indexed Sequential. VSAM, as it is commonly called, is found only on high-end mainframes. It is an implementation of indexed sequential access in which the file is also maintained as a "virtually" sequential file; i.e., it is physically sequential inasmuch as is possible, and is maintained that way by performing page splits on the file itself, similarly to the way a B-tree performs page splits in the index (see Updates below).

2.4.1.3. Versions. Versioning refers to the way in which the OS is able to access prior states of the file (not to be confused with backup - see Imaging below). In a simple OS this is typically left as a user problem. In high-end mainframes, it is the rule rather than the exception (e.g., an accounting system needs to create its monthly billing as of the end of the month but may not do so until the tenth of the following month).

2.4.1.3.1. Snapshots. The "do nothing" approach is to copy the entire file as it exists on the date for which processing is desired. This is workable for small files but virtually prohibitive for very large files (e.g., the transactions of a major bank).

2.4.1.3.2. Audit Trails. In medium to large systems, the OS can restore a file to a prior state by undoing the actions recorded in the audit trail (which it maintains for other purposes - see Imaging below) that occurred since the desired date. This allows access to current data to proceed as normal, while the program accessing a prior version will incur an added processing expense.

2.4.1.3.3. Differential Files. When accessing historical data predominates access to current data, the OS can perform updates through a "differential" file. In this approach, usually found only on high-end mainframes, the file remains static for periods of time that correspond to work cycles (days, months, etc.) and during those cycles, updates are recorded in a separate "differential" file. Every access request is accompanied by a time stamp indicating the currency of the data requested. The OS obtains the baseline information from the file and then "picks" the differential file for upd s that have occurred between the static file time and the requested file time. This has the opposite effect of audit trails in that the cost to access historical data is less, while the cost of accessing current data is correspondingly higher.

2.4.2. Device. Devices refer to the physical hardware components with which the OS must interface, including the devices on which data files reside. The principal characteristic of an OS with respect to devices is its flexibility in interfacing to different devices, either existing, new or virtual. The means by which an OS interfaces to a device is referred to as a "device driver" and it is the presence of device drivers, separately from the file system, that allows the user a high degree of uniformity in performing and redirecting [7] I/O operations and also allows the OS to unify its internal structure for most of the standard functions (e.g., open, close, read, write, etc.) performed on files.

2.4.2.1. Installed. An older OS (e.g., Unix) is typically supplied with a set of device drivers already installed as an integral part of the OS. It assumes that there is a standard complement of devices in the environment and is able to interface with exactly those devices.

2.4.2.2. Loadable. A modern OS is also supplied with a set of device drivers, but they are not installed in the OS. Instead, the user defines a "configuration" file that specifies which devices actually exist in the environment; and the OS interrogates this file to determine which drivers to load. This has the advantage of consuming only the memory necessary for the device drivers actually needed at any given time.

2.4.2.3. Definable. An OS with definable drivers goes one step beyond an OS with loadable drivers by specifying a standard protocol for defining device drivers. This allows the user to write and use drivers not supplied with the system, either to interface to a new device, but more often to create a virtual device out of existing hardware so that it can be treated uniformly as a file [8].

2.4.2.4. Raw. A "raw" device driver is a driver that allows the user to access an existing device directly, **without** going through the OS file system, while concurrently allowing the OS to use that device as part of the file system. Raw I/O is usually implemented as a partition of the device, particularly for storage devices such as disk drives. In such a case, the OS supports the definition of only a subset of the device as a standard file, and leaves the balance of the device available for either the user or another OS. This is the principle means by which applications can be easily ported from one OS to another by running the original OS as a user process of the new OS.

2.4.3. Cache. The cache of the OS refers to that part of the file system that buffers data between memory and the file. The way in which the cache is implemented is a primary characteristic of OS integrity and performance.

2.4.3.1. Synchronous. If an OS provides a synchronous cache, it means that data to be read from or written to a file is physically read or written immediately upon the user's request. This guarantees that the physical file will always be consistent with the user's image of it and conversely. This provides a very high degree of integrity and is very easy to implement. In fact, it is often the case that an OS provides a synchronous cache with no implementation at all; i.e., I/O is performed directly to and from user memory immediately upon request. However, a synchronous caching system is also the most costly, as every I/O request suspends the requesting process until the I/O is complete and may even cause other processes to be suspended if their I/O requests overlap in any way (e.g., the same device, the same channel, etc).

2.4.3.2. Read Ahead. A "read ahead" cache preserves some of the integrity of a synchronous cache in that all writes are synchronous. However, reads are anticipated and performed **before** being requested. This occurs either explicitly by the user supplying an "intent" parameter when the file is opened, or dynamically by monitoring the usage of the file. Read ahead is almost always done sequentially and is accomplished by reading one or more components of the file during a time when the user process is inactive in hopes that the components read will be requested by the user process when it is reactivated. The sacrifice in integrity is that, unless a locking scheme is employed (see below) if more than one process is accessing the same file, the image in memory that was read asynchronously for one process may not be consistent with the intentions of the other process in a way that is impossible for the OS to detect. E.g., the second process may synchronously change the file while the first is still processing based on the memory image it received; and if it makes a change to the file, it will supercede the changes made by prior change.

2.4.3.3. Write Behind. When "write behind" is added to "read ahead" caching, the result is a cache that is completely asynchronous with the processes using it. When a read request is received, the cache is first checked to see if the data is already in memory. If it is not, the request is placed on a heap with all other pending I/O requests. When a write request is received, the only effect is to set a flag in the cache image indicating that it was changed since it was read and needs to be rewritten to the file, and to place that write request on the heap. The physical I/O takes place at the leisure of the OS as follows: (a) the heap is sorted by physical address and processing commences in that sequence; (b) each read request first checks to see that there is a clean (empty or unchanged) page in the cache and if there is, the read is processed, otherwise it is placed back on the heap; (c) each write request is processed and its cache page is flagged as empty; and (d) the processing continues until either all the requests are serviced or the system is interrupted by a higher priority process. The net performance advantage is that, for a disk, the arm motion is that of an elevator, smoothly moving from the outer track to the inner and back again, depositing and picking up data as it goes. The integrity lost is that neither reads nor writes occur in the sequence in which they were requested.

2.4.4. Imaging. Imaging refers to the means by which the file system is able to recover from a failure, either of a user process, or of the media itself. It is an important characteristic of the OS in terms of both currency of data and system availability.

2.4.4.1. User. The "do nothing" approach is to allow the user access to a physically independent device during processing on which it can record anything it wishes. When a media failure occurs, it is the user's responsibility to run a process that can interpret what was written and attempt to reconstruct any lost or corrupted data.

2.4.4.2. **Before.** "Before imaging" is a feature provided by the OS that automatically writes a copy of every page of a file that is read to a physically independent device. When a failure occurs, the OS replaces every page of the file with its most current "before image" to effect a "roll-back" of the file to the most current consistent state that existed prior to the failure. Before imaging is a "pessimistic" policy that incurs a large overhead during normal operation (every read causes a write), but is able to quickly restore the file in the event of a failure.

2.4.4.3. **After.** "After imaging" is a feature provided by the OS that automatically writes a copy of every page of a file that is written to on a physically independent device. It implicitly assumes that a backup copy of the file is made at regular intervals. When a failure occurs, the backup copy is used to reinstate the file, and the latest "after image" of every page that has changed is applied to the file by the OS to effect a "roll-forward" of the file from a backup state to the most current consistent state that existed prior to the failure. After imaging is an "optimistic" policy that incurs little overhead during normal operation (only writes are duplicated and there are very few writes compared with reads), but massive overhead in the event of a failure since the entire file must be recreated from its backup copy.

2.4.4.4. Mirror. Mirror imaging is very reliable and very expensive and is employed only in systems where the ability to recover from a failure very quickly is critical. A real-time air traffic control system would have such a requirement. In mirror imaging, the OS has access to replicate hardware which, while not dedicated to the OS, is used only for lower priority functions that can be dispensed with in the event of failure. In this approach, the OS begins by creating a mirror image of the files on the replicate system. Subsequent to that, every I/O operation is written to a log on the replicate system which, periodically and quite frequently, is suspended from any other activities to process the log and bring the mirrored files up to date. When a failure occurs, the OS aborts any processes using the replicate hardware, processes any outstanding I/O operations, and resumes processing with the new hardware.

2.4.5. Update. The means by which an OS physically updates files is highly correlated to the way in which the cache operates, and whether and to what extent the OS provides imaging. However, the characteristic of importance in updating is that of performance.

2.4.5.1. In Place. In a performance intensive system, particularly when sequential processing is involved, the OS uses an "update in place" policy to preserve the physical clustering of file components. However, since updating in place may require a long update cycle (i.e., pages may have to be split and data reallocated), it is only employed in systems that have either a very robust recovery mechanism (sufficient to recover from a failure during the update cycle itself) or little or no requirement for recovery.

2.4.5.2. Replace. In systems that do not require high-performance sequential processing and in which failures are frequent and recovery support is minimal, the OS uses a "replace" policy to reduce the time during which an inconsistent file is exposed to failure. In replacement, updates never touch the existing pages of a file or its page map except for the very highest level of the page map which is usually a single page. When an update is processed, a new page is allocated from the free list for the data, and for every page of the page map that is affected. The OS then proceeds to write the updated data onto the new pages.

However, until the top-most page of the page map is written, none of the new pages are logically connected into the file, and the top-most page is the last to be written and it is only when it is written that the free list (which is always in memory) is only synchronized. Thus, the only time a failure can leave the file in a corrupted state is the brief interval when the top page of the page map and the free list are synchronized. Any failure before that interval leaves the file as it was before the update cycle began. The cost for this minimal exposure is that the pages of a file are scattered in location making sequential processing extremely expensive.

2.4.6. Lock. Locks refer to the serialization mechanism by which an OS allows multiple processes to share files without allowing either conflicting or unpredictable results. The way in which an OS locks files is an important characteristic of both its integrity and performance. It is not always the case that an OS even provides locks and it goes without saying that such systems either have little or no concurrency to deal with or little or no concern for integrity (e.g., the last process to write the record wins).

2.4.6.1. Granularity. The granularity of locks refers to the level at which locking occurs. In an OS that supports a low level of concurrency, locks are typically very coarse; e.g., an entire file or even an entire device will be unavailable to any other process until the process that locked it terminates or otherwise releases the file. To support a high degree of concurrency, the OS will provide for very fine locking at the page or even the field level so that many processes can concurrently update the same file so long as they don't attempt to change the same data value.

2.4.6.2. Exclusivity. The exclusivity of locks refers to the number of modes in which locks can be obtained and the compatibility among modes. In a simple OS with low concurrency there will be a single mode of locking that is exclusive; i.e., a process having a lock on a resource is incompatible with any other process having a lock on that resource. In an OS that supports a high degree of concurrency, there will be a large number of lock modes (e.g., read only, read with possible write, read with write intent, write intent, write for sure, etc.) and many mode compatibilities; e.g., a process having a write intent lock on a resource is compatible with any number of other processes having a read-only lock on that same resource.

2.4.6.3. Implicity. OS's will vary in the degree to which locking is implicit and to the degree to which explicit locking is permitted or required. At one extreme, an OS may require all locking to be explicit, leaving all responsibility for correct locking to the user. This has the advantage of being very fast and simple and the disadvantage of providing little guarantee for integrity. At the other extreme, an OS may perform all locking implicitly and not permit explicit locks. This has the advantage of providing very high integrity at the expense of a great deal of overhead incurred as the result of limited knowledge. An advanced OS will provide implicit locking at all times, but will also permit a user that knows he will rewrite an entire file to explicitly lock the entire file so the OS will not incur the overhead of implicitly locking it one page at a time (or of bothering to allow the process to even start until the entire file is available to be locked).

2.4.6.4. Deadlock. Any locking protocol in which the granularity is less than an entire set of related files gives rise to the occurrence of deadlock. Deadlock is a situation in which two processes are waiting for each other to release a resource [9]. The major variants involve how deadlocks are detected, and how they are broken. An elaborate OS will periodically piece together a wait-for graph and examine it for cycles (a deadlock can occur transitively among many processes). If one is found, it will analyze the progress of each process whose elimination would break the cycle and abort and restart the process with the least time invested. A simple OS will merely abort all processes at some regular interval and restart them in a random order.

2.5. IPC. Inter-process communication (IPC) is the means by which an OS allows different processes to communicate with each other. One major issue is the channel by which the communication occurs. The other is how the processes are synchronized in order for the communication to occur; i.e., how does one process know that another process wants to communicate with it. Both are important characteristics of both the performance and functionality of an OS and both present interesting problems for the OS since such a great effort is invested from preventing processes from interfering with each other.

2.5.1. Files. Perhaps the simplest approach is that of allowing the users themselves to establish a file on which messages will be written and from which messages will be read. As long as one process is the sender and the other is the receiver, the OS lock system will synchronize the processes correctly. The receiver periodically requests a read lock on the file. If it is granted, it releases the lock and proceeds. If it is refused, it upgrades its request to a write lock waits until the lock is granted and then reads the message. The sender, when it has a message to send, repeatedly requests a write lock and releases it until the lock is not granted and then re-requests the write lock and waits for it to be granted and writes the message, holds the lock for a mutually agreed upon interval (the frequency with which the receiver requests read locks) and then releases it. The result is a signal/semaphore between both processes.

2.5.2. Shared memory. Two processes can share some common memory and establish their own signal/semaphore protocol. This is faster than the file system but requires a small participation of the OS overtly since memory is normally protected from other processes. I.e., the OS must support an explicit "share" memory allocation request.

2.5.3. Pipes. The simplest completely overt support for IPC by an OS is to support an explicit request for IPC from one process as a sender and another process as a receiver. When a matching pair of requests are received, the OS establishes a "pipe" from the sender to the receiver. The pipe is a memory queue into which the sender writes and from which the receiver reads. Both sender and receiver wait on "full" and "empty" signals respectively (two-way communication requires two pipes).

2.5.4. Sockets. A more complex overt IPC support is a message pool into which many processes can write and from which many processes can read. A socket works like a pipe but augments the simple full/empty protocol with a process identifier so that messages can be "addressed to" the intended process or processes.

2.5.5. Rendezvous. The most advanced IPC is that provided in the APSE (Ada Programming Support Environment) which is, for all practical purposes, an OS in and of itself. In process rendezvous, any process can request, from the OS, a rendezvous with one or more processes. The OS coordinates all such requests and when a consistent set of requests have been received, all of the participating processes are interrupted and reactivated in a new context which is created by the OS solely for the purpose of IPC, and with a new code segment generated explicitly by the compiler for handling IPC. It is supposed to be the case that the compiler or the APSE checks that all rendezvous code is consistent.

3. LAN System Software.

Local Area Networks (LAN) are sufficiently "new" institutions that there is little standardized terminology and little agreement on even what comprises a LAN. On this question, the fundamental distinction is whether the processors (hardware systems) are closely or loosely coupled. A closely coupled LAN is a grouping of hardware systems under the control of a single LAN software system. In this configuration, the LAN is an operating system and has all the characteristics just described; and there is no issue of OS/LAN compatibility since only one system exists. This configuration will not be discussed. A loosely coupled LAN is a grouping of hardware/OS software systems that cooperate in order to achieve some higher-level common goal. In this configuration, even if the operating systems are uniform, it is of paramount importance that the LAN software and the OS software system(s) be compatible in many respects because each OS is in control of its own hardware; the LAN is in charge of distributing data, processes and control among the OS's; and each OS at any given time may operate completely independent of the LAN, completely subservient to it, or somewhere in between. It is in this loosely coupled context that the following discussion will proceed.

A LAN consists of, first and foremost, a net (or bus or channel or backplane or ...) to which separate systems are connected via a hardware interface board in the same way that they would be to any other peripheral device (i.e., the net does not have the status of the system bus unless it is explicitly accorded that status by the OS software). The net can be of any kind of media (fiber optics, coaxial, twisted pair, etc.) or combinations of media; and a net can be of any kind of topology (star, ring, drop, etc.) or combinations of topologies. Neither of these parameters are of issue. Secondly, the LAN software is software and all of it must run somewhere, but not all of it necessarily runs at the same place, and some parts of it may run at several different places concurrently. To ease the discussion, the term "service" always refers to some coherent piece of software regardless of how it is distributed among systems; and the term "server" refers to a single hardware/OS that is connected to the net which will also be referred to as a node.

In this context, the major concepts of which a LAN is comprised are servers, services, the mapping of services to servers, address protocol (how messages are sent and acknowledged among nodes), and transactions (tasks accomplished by the LAN).

3.1. **Server.** To reiterate, a server is a single hardware/OS system or "node" that is connected to the net for some reason (its participation), in some way (its coupling) with a given status (set of privileges). These, taken singularly, serve as the principal server characteristics. Taken collectively, they describe the hardware component (other than the net itself) of the LAN.

3.1.1. **Participation.** The degree to which a server is a producer for the LAN is its raison d'etre. This can range from being totally dedicated to the LAN in which case it serves no other functions independently; to being nothing but a consumer of LAN services in which case it contributes nothing and exists solely to perform other functions. Typically, a server is both a producer and consumer of LAN services to some degree and the degree to which it is a producer characterizes the server as far as the LAN is concerned; and the collective degree to which servers are producers characterizes the LAN as a whole in terms of its capacity to do work. LAN's typically have servers of all of the types mentioned.

3.1.1.1. **Dedicated Servers.** A dedicated server exists solely to serve the LAN. Dedicated servers are usually so because of their hardware resources. For example, a LAN may contain a print server which is connected to a collection of printers of various types, and exists solely as a print station to serve other nodes.

3.1.1.2. Partitioned Servers. A partitioned server is usually a multi-processing node that is partitioned so that some of the processes are dedicated to the LAN, while others are invoked on behalf of users of that node. An exemplar of such a node would be main-frame system that supports a large database for its users, but some of that database is public to the LAN. In this case, the OS of that server might create some primed processes to service incoming LAN requests, and others to service its own users.

3.1.1.3. Available Servers. An available server is a node which is functionally dedicated to its users, not the LAN, but at varying times (e.g., at night) has excess processing capacity which it exports to the LAN. To put it another way, when the LAN workload is sufficiently heavy, the LAN software may solicit help from nodes that are connected to but not functioning on behalf of the LAN at the moment.

3.1.2. Coupling. The coupling of a server to the net occurs through a hardware interface board or "net board." What is important about the net board in terms of characterizing the server and the LAN as a whole are: (1) the functions of which the net board is capable; and (2), the presentation of the net board to the balance of the server.

3.1.2.1. Board Functions. A net board may be very simple, capable of merely translating a byte from an interface register to a signal compatible with the net and conversely. At a higher level of functionality, a net board may implement the entire address protocol so that all the node has to do is create a message in a transmit buffer whose location is known to the board, and the board will carry out all the functions necessary until a receive buffer whose location is known to the board has been filled with the desired response. In addition, depending on the topology of the net, numerous LAN functions may be carried out directly by the board without interrupting the node at all. Such functionality might be as a repeater in a ring or a reflector on a drop.

3.1.2.2. Board Presentation. The way in which the net board appears to the server can vary widely. It may be interfaced as a peripheral port; i.e., a 10 bit register with 8 data bits and two status bits. In this case it will appear as a device and will probably be accommodated by the OS as part of the file system through an associated device driver. By contrast, it may be mapped directly into one or more memory pages it which case it may be accommodated as either a process that is accorded a share of the execution cycle of the node, or as shared memory that appears to be an IPC area. When the latter is the case; i.e., the board appears as IPC; there is usually one or more OS processes communicating through it to OS processes on other nodes. Such a presentation is called "message passing" and the OS processes are called the "message passing kernel" (see below).

3.1.3. Status. Every server acquires one or more statuses with respect to the LAN and requires at least the status of "being known" to at least one other server in the LAN before it can acquire any additional status. The initial status assignment can be made in a variety of ways depending on what other statuses are defined for the LAN. I.e., the initial status assignment protocol, alone, is a highly characteristic of the LAN in terms of status.

3.1.3.1. Existence. Presuming that an "about to be" node has the necessary hardware and software that enable communication with other nodes of the LAN, some process must occur by which the node becomes "known" to these nodes. This procedure will vary in two principle dimensions based upon the mode of addressing employed by the

LAN, and the way in which statuses are acquired within the LAN. Only the latter is of issue here.

3.1.3.1.1. **Hierarchic .** In LAN in which statuses are hierarchic, a new node will typically become known by virtue of an action on the part of the highest level node or "net manager." I.e., an end user with an equivalent status (e.g., the LAN administrator) will enter the name, physical LAN address, resources, and other descriptive information about the node, and grant it privileges and possibly responsibilities with respect to the LAN.

3.1.3.1.2. **Collateral.** If the LAN is primarily collateral; i.e., all nodes are essentially peers; a new node will typically become known by one of two processes. If the LAN implements security, the new node must be "escorted" into the LAN by another node which will, on behalf of the new node, introduce it to that compartment of the LAN in which it is known and it will acquire the privileges accorded that compartment. If the LAN has no security, the node may log onto the LAN as a consumer terminal, and request admission to the LAN. This request will be fielded by some service of the LAN that deals with incorporating new nodes.

3.1.3.2. LAN. Having LAN status carries the privilege of being solely in charge of the LAN and directing the behavior of any other server. Such status is accorded under one of two circumstances. If the LAN depends on having a "master/slave" relationship, some server will be accorded LAN status either manually, when the NET cold--starts [10], automatically when the server with LAN status elects to abdicate this status to another server, or after a failure of the LAN-high server. If the LAN is collateral, there is, by definition, no such status defined. There will be, however, a similar set of privileges accorded to a "Net Maintenance Service (NMS)" whose primary job is to run diagnostics and correct failures. Every server is obligated to cooperate with and obey the instructions of the NMS if the LAN is to be durable, but the NMS is primarily a passive function and will only take on a leadership role when there is a failure of the LAN and will dispose of that role as soon as possible thereafter.

3.1.3.3. Transaction. In collateral LAN's, leadership is accomplished on a task basis. To be successful across failures, every transaction undertaken by the LAN must be managed by a single server, with all other participating servers subservient to it with respect to that transaction. Note that, explicitly, a transaction only involves multiple services, not servers. However, the mapping of services to servers will usually yield the case of multiple servers participating in a transaction. Nevertheless, it is not sufficient to the transaction management protocol (see below) that a single service be in charge (that is always the case anyway since transaction management, itself, is a service), but rather that a single server be in charge, and it will likely be (but not necessarily) one of the servers invoked by one of the services, and usually the server in which the transaction began. The result is that, at any given time, a server will concurrently have the status of transaction manager for some transactions as well as being a participant in other transactions and being subservient to those transaction managers.

3.1.3.4. **Service.** If a service is distributed across multiple servers, one of those servers may acquire the status of a service manager. This is not a requirement, but rather the fallout of the design of the service. For example, a database service may well be designed to have a single server fielding all requests for the service and parsing the requests prior to distributing them so that parse trees are the means of communicating and parsing does not have to be repeated. Since that server is the node that acknowledges the receipt of all requests, it is reasonable to have that node be in charge of all request scheduling and result distribution which effectively makes it a service manager.

3.1.3.5. **Sub-service.** If there is a service manager, then other servers in that service may take on a sub-service status which is to say that they may not even be known to servers outside the service, and respond only to requests of the service manager. Again, this is usually a service design consideration.

3.2. **Service.** A service is a functionally coherent software system that is capable of performing one or more well-defined tasks for the LAN or its users. Some services are integral to the LAN and the LAN could not operate without them. These would include net maintenance, transaction management, addressing, etc. Others are much like general applications such as word processing and file management while still others are organizational applications (e.g., accounting). For the purposes of this discussion, it is not as important to pursue the question of what services exist as it is to discuss what kind of services can exist for this will characterize the LAN as a general purpose system. And the principal characteristic that limits the possibility of a service is the mapping of services to servers.

3.2.1. **Confined.** A service is said to be "confined" if it must exist on a specified server. Confined services arise either because of specific hardware requirements or a limitation of the LAN. An exemplar of the former would be a LAN in which only one server had a printer. The print service would obviously be confined to that server. The latter occurs when the addressing functionality of the LAN is limited such that: (1) the only status conferrable to a server is that of service manager; i.e., there is no sub-service status; and (b) a service is addressable only by name and cannot be qualified by location.

3.2.2. **Migrant.** A migrant service obeys the constraint of a confined service that it must exist on a single server; however it is not necessary to specify which server. It is thus able to migrate from one server to another but will only exist on a single server at any given time. Migrant services could arise from a LAN limitation that a service cannot be distributed or replicated (see below) but usually arise because of a functional requirement of the service. An exemplar of the latter would be the net maintenance service. Even if the LAN supported replicated services, the net maintenance service would not be allowed to exist on multiple servers because they would be in mutual conflict.

3.2.3. **Singular.** A singular service refers directly to the LAN constraint that messages are addressed to services by server name through a 1:1 service:server mapping. Under such limitations, the only way to replicate a service on more than one server is to give it another name. E.g., if it is desired to have the print service at two locations, the services would have their location embedded in their name such as Print-LRI and Print-IAS. The LAN would not be aware that the services had any similarities and would never attempt to use one in place of the other.

3.2.4. **Plural.** A plural service is permitted when LAN addresses contain a location component. Each instantiation of the service is still restricted to be wholly contained within a single server, however there may be many such servers offering that service. For example, if many of the servers in the LAN have printers, then all of those could run the print service concurrently and a user could either specify a particular location at which the printout is desired, or could allow the print service to schedule the print request on the first available server. When plural services are permitted, it is necessary that something schedule and route requests for the service. In some cases, particularly those in which scheduling does not require knowledge of server resources, there may exist a scheduling service that will route the request. When inside knowledge is required (e.g., in a file service where each server has different files) the service will normally provide its for its own scheduling through a message protocol.

3.2.5. Distributed. A distributed service is one in which the service spans several servers, each running only part of the service. The ability to have a distributed service is not limited by the LAN directly, but is more or less facilitated by the LAN through various support mechanisms. At the least level of support, the designers of a distributed service will have to determine a priori how to allocate the service among servers, provide for their own intra-service message and transaction protocol, and to decide the means by which requests for the service will be fielded and the results disseminated. At the highest level of support, the LAN will provide all of these functions as services and will provide a distributed compiler that allows the user to write the service without concern for which procedures reside on different servers and which are collocated, a fact which is, itself, one that can change dynamically during processing.

3.3. Address. For one service to address another service is to get information to it (a request) and receive information from it (a response) and this represents the inter-service communication capability of a LAN. In addition, if the LAN supports distributed services, it is also necessary for the processes of a service to address other processes of the service which may not be collocated on the same server and this represents the intra-service communication capability of the LAN. It is possible for a LAN to employ different mechanisms for inter- and intra-service communication and such will be the case where there is a vast difference in the volume and frequency of messages of each type. Conversely, the LAN is considerably simplified if only one form of address serves both requirements. The way in which a LAN supports address is characteristic of its functionality, its performance, and its utility as a whole. The following descriptions assume, for simplicity, that all nodes are connected directly to the net, although this is not a requirement (connections can be virtual even though they may not physically exist at the lower levels of the net protocol).

3.3.1. **Broadcast.** By far the simplest means of synchronous address is broadcast. In broadcast addressing, the sending service encodes a message with both a source and destination identifier and transfers it to the net board for transmission. The net board then repeatedly transmits the message each time it receives the token (in a token-passing net) or finds an available slot (in a slotted net) for some routine interval. The receiver side of the board both downloads every message on the net into a receive buffer as well as repeating it if the net technology requires repeaters. Conversely, the net board examines every message on the net and determines whether the encoded address matches (or includes) its own address. If so, it removes the message from the buffer, replaces it with an acknowledgement message, transmits the acknowledgement message and generates an interrupt in the server. Whatever service is running in the server proceeds to examine the message and determine what, if any, action it should take. As would be expected, broadcast works well when services are highly replicated resulting in a high ratio of receivers to senders, messages are highly asynchronous, and security is of no concern.

3.3.2. Circuits. Circuits are a refinement of broadcast addressing in that only the source and destination identifiers are broadcast. The receivers, when they get a matching address, then prepare to receive the message contents as the very next message with a matching identifier. The sender waits for an acknowledgement from all destinations, if all destinations are known, or for some period of time otherwise and then broadcasts the content of the message the next time it has access to the net. Circuits greatly reduce both the amount of information on the net and the amount of time spent downloading and examining information, especially if messages tend to be quite large.

3.3.3. **Packets.** Packets are a refinement of circuits that allow messages to be broken up into short segments so that only part of the message is transferred each time the circuit is established. Packets provide for a fairer distribution of the net in the presence of long messages while still preserving message synchrony. When messages are typically very short, the difference between circuits and packets becomes minimal.

3.3.4. **Datagrams.** A datagram refers to a message prefaced by a source and destination address that is sent independently of any other messages flowing between those addresses. As such, it is very simple to implement but does not encode any form of synchronization. I.e., datagrams are not necessarily received in the order they are sent and there is no guarantee that a datagram will be delivered. Datagrams are very efficient for inter-service communication which is usually of an asynchronous nature anyway, and for intra-service communication in which the synchronization is so complex that the service will normally implement its own synchronization protocol regardless of what the LAN provides (transaction management service is of this type). It is quite common for a LAN to be partitioned with some of the bandwidth reserved for circuits, and any time unused by circuits available for datagram traffic.

3.3.5. Message Passing. Message passing refers to the transfer of data directly between or among the processes that implement a service via an underlying inter-process communication service. Such a service is referred to as a "message passing kernel" and has the distinct advantage that the processes of a service communicate with each other at all times as if they were running on the same server. Thus, programs written to provide a function in a stand-alone environment are very easily ported to provide those functions as services on the LAN. The difficulty with message passing is that the user of the kernel implicitly depends on the kernel to provide the same level of synchrony and guarantee that the OS IPC does. It also requires that a context switch occur in the OS each time the kernel is accessed since it runs as an OS process; i.e., each IPC, in effect, results in three IPC's, one from the local process to the local message passing kernel, another to the remote kernel, and a third between the remote kernel and the remote process.

3.3.6. Tickets. Tickets are an advancement over message passing that bear the same relationship to it as primed processes do to forked processes. When a receiving process initializes for a processing session, it broadcasts (via the message passing kernel) a set of "blank order forms" called tickets in the amount of the number of requests it expects to service during that session. Other services similarly retrieve as many tickets as they expect to need during that session. This has the effect of pre-establishing all of the IPC's to the point of being ready to accept data (similar to a virtual circuit). When a requesting service has an actual service request, it simply fills in the contents of the message which causes the service that sent the tickets to be interrupted.

3.4. Transaction. The purpose of a LAN is to accomplish work by a concerted and cooperative effort among the services of the LAN. To do so in an orderly manner, the LAN implements the concept of a transaction to represent a unit of work in much the same way that an OS implements the concept of a process. A transaction has three basic properties: it is atomic (i.e., either it occurs or it doesn't); it is durable (i.e., once completed its effects can only be altered by another transaction); and it is consistent (i.e., if the state of the LAN was consistent before the transaction it will be so after the transaction). The means by which the transaction concept is implemented in the LAN is the primary characteristic of its integrity. The function of transaction management may be implemented as a service or directly in the kernel of the LAN.

3.4.1. **Implicity.** The degree to which transactions are explicitly started by the user or implicitly started by the LAN has a direct bearing on its integrity and performance. Some LAN's will implicitly begin a transaction at the first attempt to access a recoverable resource [11] by any process. This offers high integrity when it works correctly but this is not always the case since the LAN does not usually know which other processes the accessing process is cooperating and must infer this from the IPC. The opposite extreme is to provide for an explicit Begin Transaction call. If the protocol is followed by all users, the integrity is just as high if not higher than implicit transactions. Explicit transactions also allow the user option of **not** invoking transaction management if the user in unconcerned about the aftermath of a failure; or knows it will not engage in any actions that could leave the system in an inconsistent state.

3.4.2. **Serialization.** In order to implement transactions, there must be a mechanism for serializing the actions affecting recoverable resources so that if a failure occurs, the sequence of actions taken or intended can be reconstructed and either re-done or un-done. The means by which a LAN implements serialization is a principle characteristic of the LAN and the possibilities for serialization are as follows.

3.4.2.1. Locks. Locks inherently sequence the processes accessing resources through queues. The implementation of locks is similar to that described for the OS file system. Locks are successful if all users obey the locking protocol. Failure to obey the protocol does not allow a user to actually access the resource, but it denies the LAN of the knowledge of the sequence in which requests for the resource occurred.

3.4.2.2. Timestamps. An alternative to locks are timestamps. Every process request to access a resource is appended with a timestamp indicating "when" the request was issued. A timestamp is not usually derived from a clock [12] but from a counter that is passed among the servers by the LAN. When a server receives the counter, it increments it for each resource request it issues and then passes it to the next server. A timestamp is similar to a token in a token-passing net. Timestamps are relatively expensive to implement since they require that every server update and pass the clock even if that server has no resource requests (which is most of the time). They also impose the requirement for a very elaborate restart process (see below) in the event that the failure causes the clock to be lost (e.g., if the failure occurs in the server with the clock).

3.4.2.3. Tickets. Tickets are an extension of tickets already described for messages to cover all resources of the LAN. In ticket serialization, a process acquires numbered tickets for each resource it expects to request during a processing session. When it has a request, it issues the request appended with the lowest numbered ticket in its possession. If the request is not granted, it continually reissues the request appended with the next highest ticket until the request is granted. Tickets "pre--queue" resource requests in order to absorb as much of the serialization overhead as possible in advance.

3.4.3. Logs. If recovery is to be possible, every transaction must create a log of all its actions (or intentions - see abort processing below) with respect to each recoverable resource which it accesses during the transaction. Note that, in order to be of any use, a log must always be treated as a **non-recoverable** resource; i.e., it is the very fact that the log is inconsistent after a failure that makes it possible to recover other resources to a consistent state. The kind of log maintained is characteristic of the LAN in both performance and integrity.

3.4.3.1. Undo List. An undo list is an optimistic approach that assumes that most transactions complete successfully. The LAN, when it receives a request to change the state of (update) a resource, writes the current state of the resource to the list and then actually services the request. It is optimistic in that, if the transaction completes successfully, nothing else has to be done; the updates have already been processed. In the event of a failure, the prior state of the resource is recovered from the list and the resource is synchronized to that state.

3.4.3.2. Intentions List. An intentions list is a pessimistic approach that assumes that most transactions fail before completion. The LAN, when it receives a request the change the state of a resource, writes that request to the intentions list and does nothing to the resource. When the transaction completes successfully, the LAN reads the requests from the intentions list and services them at that time. It is pessimistic in that when a failure occurs, nothing else has to be done; no updates to resources have actually occurred so the resources are still in the state they had prior to the commencement of the transaction.

3.4.3.3. **Write Ahead Log.** A write ahead log (WAL) is an approach that assumes that the processes issuing requests are the only things capable of correctly recovering resources. The LAN, when it receives a request to change the state of a resource, returns a unique log address to the calling process. It then holds the request in abeyance until it receives a write request to the log at that address at which time it commences to service the original request. The LAN has no knowledge of what is written to the log except the identifiers of both the process and the resource and the log address. When a transaction completes either successfully or unsuccessfully, the LAN recalls every process and passes them the list of log addresses. What the processes do when they are recalled is unknown to the LAN. Transactions implemented via WAL are called two-phase transactions; i.e., every request is processed twice, once during phase one of the transaction (prior to destiny), and again during phase two (when it is known whether the transaction was successful or not).

3.4.4. **Coordination.** Whenever a transaction involves more than one process the processes have to be coordinated during the transaction. Moreover, if the transaction involves more than one server, the coordination must be distributed. Coordination may be implemented by the transaction management service directly as part of an existing process, as a new process, or as a set of processes. If the transaction is distributed, the latter of these will be a requirement since a process cannot span servers and this will be the case assumed in the following discussion. The functions of transaction coordination that characterize the LAN are the way in which the transaction is initiated, how processes enter and exit the transaction including the effects of nesting, and how the transaction is concluded (either committed or aborted).

3.4.4.1. Begin Transaction. When a transaction begins, either explicitly or implicitly, it must be given a unique identity and some structure must be established for recording information about the state of the transaction (similar to what an OS must do when a process begins). This is accomplished by establishing a process when the transaction begins to serve as a place holder for the transaction. The process may be established on the server where the transaction began, the server where the (first) recoverable resource is requested, or a server dedicated to transaction management service. The former approach is preferred for performance in that no messages will be generated unless the transaction is distributed. The latter is preferred for simplicity but incurs a very high risk for the LAN as a whole if a failure occurs in that server. In all cases, the process that requested the transaction is given the unique transaction identifier which will be appended explicitly or implicitly to every request of that process until the transaction completes.

3.4.4.2. **Entry.** A process enters a transaction whenever it is invoked by a process that is part of that transaction. This will normally be explicit to the invoked process to the degree that the request to begin the transaction by the invoking process was explicit [13]. If the entering process is on the same server, the transaction management process on that server needs to be informed about the new process. This can be accomplished by an explicit request by the invoked process to enter the transaction, or the transaction management process can be invoked by the LAN automatically as part of process invocation sequence. In the latter case, the OS on that server must, at a minimum, be aware of the transaction through some mechanism. If the entering process is on another server, then the transaction management service must spawn a process on that server to either parallel its own functioning, or to intercept and pass transaction related messages. It is not necessary for the user to know which method is employed; and in either case the remote transaction management process will be invoked subservient to the original transaction process (i.e., will not only run as a part of that transaction, but will have IPC with it).

3.4.4.3. **Nesting.** When a process enters a transaction, the question arises as to whether that process is nested in the transaction in the same sense as it is nested in the process that invoked the transaction, or it is collateral with all processes in the transaction. In the former case, the nested process will be invoked for commit/-abort processing (see below) when it terminates. In the latter case it will be invoked when the transaction terminates. If the process is nested, then its commit/abort processing must be based on its own destiny since the destiny of the transaction is unknown. If the LAN supports such nesting, it is critical that the LAN or the users or both have some mechanism for creating compensating transactions to undo the effects of a nested process that commits since the transaction may abort at a later time.

3.4.4.4. Exit. When a process exits a transaction the transaction management service must be invoked to perform exit processing. If the transaction is single-phase, this will consist of acquiring all of locks owned by the process if locks are employed, and all of the resources acquired by the process, so that the log can be processed when the transaction ends. If the transaction is two-phase, this will simply consist of noting on the log that the process has completed phase-one processing and is prepared to be recalled. In the latter case, it is necessary that the exiting process be prepared to go either way (commit or abort) at the end of phase one.

3.4.4.5. Destiny. The destiny of a transaction is either commit (successful) or abort (unsuccessful). The means by which the transaction management service determines and acts on the destiny is an important characteristic as is when the determinations and actions occur. In one phase transactions, each process is asked for a destiny when it terminates. In two phase transactions, processes will be asked for a destiny by an explicit request of the transaction management service. The determination of destiny is almost always based on unanimous commit (i.e., either all processes vote commit in which case the destiny is commit or one process votes abort in which case the destiny is abort) although some LAN's will employ a non--unanimous voting scheme. In unanimous commit, there also exists the possibility for unilateral abort; i.e., since one abort is sufficient to abort the transaction, the transaction management service has the option to interrupt all processes and direct them to commence abort processing as soon as one process terminates with an abort destiny. This is always possible in a one phase transaction by simply requesting the OS on each server to abort the processes. In two phase transactions, each process must have code to support early abort processing.

3.4.4.6. **Phase Transition.** In two phase transaction processing, phase transition refers to the processing that occurs between the determination of destiny and the acknowledgement by all processes of receipt of destiny. Once all processes have voted and the destiny determined, the transaction management service enters the single most critical stage. It broadcasts the destiny to each process and logs the acknowledgements as they arrive. It does not write the destiny (transition) record to the log until every process has correctly acknowledged the destiny broadcast. The reason for this is that a process in a transaction with a commit destiny may still fail before acknowledging the destiny which will cause the destiny to be changed. When the destiny acknowledgements have been received (or the transaction management service has timed--out waiting for one or more acknowledgements) the destiny record is written to the log and the moment that the destiny record is written is called the "instant of commit" in that any failure prior to that is equivalent to an abort, and once the record is written, the destiny of the transaction is guaranteed (see restart below). The only window for failure of the transaction management serviced is the time during which the transaction record is being written.

3.4.4.7. Commit/Abort Processing. In one phase transactions, the transaction management service commences to process the logs of all participating processes in whichever manner the logging protocol and destiny prescribe. In two phase transactions, the transaction management service proceeds to **persistently** recall each participating process to perform its own commit/abort processing and records the acknowledgement of each process that its phase two processing is complete. The calls are persistent in that a process may complete phase two processing but a failure may occur in the process or elsewhere that prevents the transaction management service from receiving the acknowledgement. It is therefore necessary that all processes in a two phase transaction be capable of receiving idempotent requests for phase two processing.

3.4.4.8. End Transaction. When the transaction management service completes log processing in a one phase transaction or receives acknowledgement from all participating processes in a two phase transaction that phase two processing is complete, either an explicit "end transaction" record is written to the log if the log is incremental, or the log is deleted. In addition, all resources and locks acquired during the transaction are released.

3.4.5. Restart Processing. The ability of a LAN to restart either after a complete shutdown or a failure of one or more components is the principle characteristic of its robustness. In a simple LAN, no explicit restart processing is performed except the deletion of any incomplete transaction logs. This results indirectly in all transactions that were in progress acquiring the status of having been aborted prior to the failure. The next level of sophistication is to examine the logs of any transactions that were in process to determine which process started the transaction and to recover the message which caused the process to be invoked. Then the log is deleted and the process is restarted with the original message. Note that this form of restart will not necessarily produce the same outcome had the transaction completed prior to failure since there is no determination of how far the transaction had progressed (e.g., messages may have been sent to other processes that have not failed). The most sophisticated restart processing involves a detailed examination of the log which proceeds as follows.

3.4.5.1. ET Record. If an ET record is found in the log, it is assumed that the transaction was completed prior to the failure and was successfully committed or aborted as indicated in the destiny record. If incremental logging is not in use, the log is deleted.

3.4.5.2. **Transition Record.** If the ET record is not present but the phase transition record is present, then the restart processor compares the list of destiny acknowledgements with the list of phase two completion records and persistently recalls any processes that are on the first list but not on the second. Again, processes are assumed to be capable of fielding idempotent calls.

3.4.5.3. **Destiny Acknowledgement.** If the transition record is not present but there is at least one vote request acknowledgement, the restart processor compares the list of vote request acknowledgements to the list of destiny acknowledgements and re-transmits the destiny to any processes on the first list that are not on the second.

3.4.5.4. **Vote Request Acknowledgement .** If there is at least one vote request acknowledgement, the restart processor re-broadcasts the vote request. The processes that did send a vote request acknowledgement before the failure will merely interpret this as a non-acknowledgement of the message and will re-transmit it. In this case, the restart processor must be able to deal with idempotent acknowledgements.

3.4.5.5. No Vote Request Acknowledgement. If there are no vote request acknowledgements, the restart processor will assume that no vote request was ever received, hence the transaction had never completed phase one and will treat it as a case of unilateral abort by broadcasting an explicit abort destiny and resuming from that point. This serves to cause processes that are still waiting for a vote request to commence abort processing, thereby cleaning up the remnants of the transaction. When the transaction reaches normal ET-abort, the restart processor will restart the entire transaction. Depending on the nature of the failure, this process may fail repeatedly to the point that some manual intervention will be required [14].

4. References.

[1]. End users may perceive it to be the case that they do, on occasion, interface with the operating system (e.g., running a program, copying a file, etc.) but this is incorrect. In such circumstances they are, in fact, interfacing with an application program commonly known as a "shell" or "command interpreter" which is totally unknown to the operating system. To reify this point, in MS-DOS a command line interpreter named "command.com" is supplied by Microsoft on the distribution disk. However, there is a variable in the MS-DOS environment table that specifies what program to execute by default if no other program is running. That variable comes initialized as "command.com" but can be changed to specify any program whatsoever such as a text editor, word-processor or windowing system. And it is these, not the OS, that the user interacts with. The user per se of the OS is the person that writes these programs.

[2]. Similarly, the problem of infinite regression in the hardware is also solved by a bootstrap. In this case, the goal is to load the operating system. But that can't happen until a process that loads other processes, a "loader" has started. And since a full-featured loader is, itself, a complex program, yet another, much simpler program is needed to load the loader. This regression continues until a very simple program (e.g., read the contents of track 0 into memory starting at location 0 and begin executing instructions at location 0) is reached. This program is then implemented in the hardware as a Read Only Memory (ROM) chip or as a set of binary switches. The latter allows for changing the bootstrap if necessary while the former does not.

[3]. Whether or not code is re-entrant is as much a function of the programmer and compiler as it is of the OS. Re-entrant code has the property that its execution can be interrupted at any point, and re-entered at any point with a different data segment. I.e., the compiler (or the programmer) provides for the capability to save all of the information necessary to restart processing after an interrupt, including the information about what values were contained in all of the data variables. Most modern compilers can generate such code with relative ease since they are stack-oriented; i.e., they push an activation record onto a stack each time a procedure begins execution, including the variables that the procedure was called with. Making such code re-entrant merely requires having multiple stacks and stack pointers. Older, unstructured languages are difficult and often impossible to make re-entrant. To

my knowledge, there are no re-entrant FORTRAN compilers in existence.

[4]. There are yet faster memories such as the instruction pipeline or pre-fetch queue for parallel or quasi-parallel CPU's respectively, but these are not normally visible to the OS. There are also hardware registers for specific CPU functions that may be visible to the OS for communicating with the CPU, but it is the CPU, not the OS, that manages the allocation of these. Unfortunately (for this discussion) there are also non-specific hardware registers that are **intended** to be visible to the OS and even to the user, but that is one level of detail too deep for this discussion.

[5]. It should come as no surprise that despite the elegance and portability of layered software (e.g., Open System Interconnect), system developers remain very reluctant to separate functions and allocate them to different processes. Their reluctance is based **solely** on the cost of performing a context switch which would otherwise be unnecessary if the functions were collocated in the same process. And until the cost of context switching is brought down to an acceptable level, monolithic operating systems will continue to enjoy a large advocacy in spite of all their undesirable consequences.

[6]. The use of the term "page" is somewhat arbitrary. Disks, for example have "sectors" and tapes have "blocks." Page is a useful term in that it is both neutral to media and consistent with the terminology of memory. And since virtual systems, do not make a strong distinction between files and memory, the consistency is desireable.

[7]. Redirecting refers to the capability of writing a program that performs I/O without binding that program to a particular device. In this case, device binding is done at process invocation rather than at program compilation and such programs are traditionally called "filters" in that they can be inserted in a "pipeline" of processes in which the output of one process is the input to another.

[8]. A primary exemplar of a virtual device is a RAM--disk. In many systems, there is considerably more RAM present than the OS is capable of managing, due to limitations of the address bus. While a user can access this additional RAM in any way he chooses, an easy way to access it is as a file that just happens to be very fast. This can be done by writing a device driver. In most systems, a driver consists simply of a standard set of routines that implement the standard file functions of open, close, get byte, and put byte, and the code to both initialize the device and turn it off.

[9]. For example, process P1 obtains a read lock on records Ra and Rb and decides based on the contents of Rb that it wants to upgrade its lock on Ra to a write lock (a write lock is incompatible with any other lock mode). Concurrently, process P2 obtains a read lock on records Rb and Ra and decides based on the contents of Ra that it wants to upgrade its lock on Rb to a write lock. P1 is now waiting for the read lock on Ra to be released while P2 is waiting the read lock on Rb to be released and neither process is able to proceed (P1 and P2 do not know of each others existence, let alone what locks they have).

[10]. The traditional terminology of "cold" and "warm" starts is misleading for a LAN. With so many processors involved, a cold or power-off start may only occur once in the life of the LAN; i.e., when it is first created. From then on there will always be some complement of servers in operation and/or some transactions in progress. Subsequent equivalents of cold and warm starts are referred to, in a LAN, as restarts; i.e., attempts to restore the net to a current operational status.

[11]. A recoverable resource is one that is guaranteed by the LAN to survive a failure of any or all of the servers in the LAN; and is guaranteed to be in a consistent state at all times in which the LAN is in a consistent state. Transactions are only meaningful in the context of recoverable resources.

[12]. The use of the term "clock" is purely mnemonic. In a high-speed LAN the use of a real clock on each server is impossible for at least two reasons: (1) the frequency of the clock would need to be in nanoseconds; and (2) since each server has its own clock, there is no way to guarantee that is in synchrony with the other clocks to the nanosecond level other than to pass some kind of synchronizing message around the LAN. And as long as a message needs to be passed, the message itself can serve as a clock without need to reference any real clock.

[13]. Unless WAL is employed as the logging mechanism, processes do not necessarily have to be aware that they are participating in a transaction. However, it is routine to provide that visibility so that the user can write programs more efficiently. When a WAL protocol is employed, all programs must provide code segments for phase 1 and phase 2 processing; a requirement usually enforced by the compiler.

[14]. The manual intervention is critical. An example would be an air traffic control system that has responded affirmatively to a request for permission to land from an airliner, but due to some hardware malfunction, failed just

one instruction prior to the instruction that would physically transmit the message. To every process in the system, and every person observing the system, it would appear that the message has been sent, even though the system is repeatedly restarting that transaction. However, unless someone manually inspects the log, the airliner will run out of fuel and crash because the only way of knowing that the message going out is not being sent is to see that, on each restart, the process that physically sends messages does not acknowledge the vote request meaning it is failing somewhere between acknowledging the end of phase one processing and receiving the vote request.

A Knowledge Dictionary System for Scheduling Support

P.G. Ossorio and L.S. Schneider

Appendix C

Submitted by
Linguistic Research Institute, Inc.
5600 Arapahoe Avenue
Boulder, Colorado 80303

Submitted to
Rome Air Development Center
Griffiss AFB, New York

Table: ..\..\data\oslan.asc Page: 1 Strip: 1

—— OS Parameters / LAN Parameters	—— Server »»	—— Participate »»
1 Process	[Exists At]	[Implements]
1a Initiate	[Occurs At]	[Initiates]
1a(1) Spawned	[Spawns At]	[Spawns]
1a(2) Forked	[Forks At]	[Forks]
1a(3) Primed	[Primes At]	[Primes]
1b Execute	[Occurs At]	
1b(1) Dedicated		
1b(2) Sliced		
1b(3) Interrupt		
1b(4) Stacked		
1c Code Segment	[Exists At]	
1c(1) Duplicate		
1c(2) Single Reentrant		
1c(3) Multiple Reentrant		
1d Data Segment	[Exists At]	
1d(1) Process-Bound		
1d(2) Code-Bound		
2 Memory	[Exists At]	
2a Real	[Part Of]	
2a(1) Linear		
2a(2) Segmented		
2a(3) Paged		
2a(4) Protected		
2b Virtual	[Exists At]	
2b(1) Swap		
2b(2) Demand Page		
2b(3) Explicit		
3 Context	[Exists At]	
3a Create		
3b Switch		
3c Exit		
4 Files	[Exists At]	
4a Data	[Exists At]	
4a(1) Format	[Exists At]	
4a(1)a) Stream		
4a(1)b) Text		
4a(1)c) Paged		
4a(1)d) Structured		
4a(2) Access	[Occurs At]	
4a(2)a) Sequential		
4a(2)b) Direct		
4a(2)c) Indexed		
4a(2)d) Indexed Sequential		
4a(2)e) Virtual Indexed Sequential		
4a(3) Versions	[Exists At]	
4a(3)a) Snapshots		
4a(3)b) Audit Trails		
4a(3)c) Differential Files		
4b Device	[Part Of]	

Table: ..\..\data\oslan.asc Page: 2 Strip: 1

— OS Parameters / LAN Parameters	Server »»	Participate »»
4b(3) Definable		
4b(4) Raw		
4c Caching	[Occurs At]	
4c(1) Synchronous		
4c(2) Read Ahead		
4c(3) Write Behind		
4d Imaging	[Occurs At]	
4d(1) User		
4d(2) Before		
4d(3) After		
4d(4) Mirror		
4e Update	[Occurs At]	
4e(1) In Place		
4e(2) Replace		
4f Locking	[Occurs At]	
4f(1) Granularity		
4f(2) Exclusivity		
4f(3) Implicity		
4f(4) Deadlock		
5 IPC	[Occurs At]	
5a Files		
5b Shared Memory		
5c Pipes		
5d Sockets		
5e Rendezvous		

Table: ..\..\data\oslan.asc Page: 1 Strip: 2

Dedicated	Partitioned	Available	Server >>
			[Exists At] [Occurs At] [Spawns At] [Forks At] [Primes At]

Table: ..\..\data\oslan.asc Page: 2 Strip: 2

Dedicated	Partitioned	Available	Server »»

Table: ..\..\data\oslan.asc Page: 1 Strip: 3

Coupled »»	Function »»	Transceive	Repeat

Table: ..\..\data\oslan.asc Page: 2 Strip: 3

Coupled »»	Function »»	Transceive	Repeat

Table: ..\..\data\oslan.asc Page: 1 Strip: 4

Buffer	Decode	Server »»	Coupled »»
		[Exists At] [Occurs At] [Spawns At] [Forks At] [Primes At]	

Table: ..\..\data\oslan.asc Page: 2 Strip: 4

Buffer	Decode	Server »»	Coupled »»

Table: ..\..\data\oslan.asc Page: 1 Strip: 5

Interface »»	Port	Device	Memory

Table: ..\..\data\oslan.asc Page: 2 Strip: 5

Interface »»	Port	Device	Memory

Table: ..\..\data\oslan.asc Page: 1 Strip: 6

Server »»	Status »»	Exist »»	Hierarchy
[Exists At]	[Implements]	[Implements]	
[Occurs At]	[Initializes]	[Initializes]	
[Spawns At]	[Spawns]	[Spawns]	[Incompatible]
[Forks At]	[Forks]	[Forks]	[Compatible]
[Primes At]	[NA]	[NA]	[NA]

Table: ..\..\data\oslan.asc Page: 2 Strip: 6

Server »»	Status »»	Exist »»	Hierarchy

Table: ..\..\data\oslan.asc Page: 1 Strip: 7

Collateral	Server »»	Status »»	LAN
	[Exists At]	[Implements]	
	[Occurs At]	[Acquires]	
[Compatible]	[Spawns At]	[Spawns]	[OS = LAN]
[Incompatible]	[Forks At]	[Forks]	[Incompatible]
[NA]	[Primes At]	[Primes]	[OS = LAN]

Table: ..\..\data\oslan.asc Page: 2 Strip: 7

Collateral	Server »»	Status »»	LAN

Table: ..\..\data\oslan.asc Page: 1 Strip: 8

Transaction	Service	Subservice	Service »»
			[Implements]
			[Initiates]
[Compatible]	[Compatible]	[OS = Service]	[Compatible]
[Nested in Parent]	[Incompatible]	[Parent = Service]	[Compatible]
[Compatible]	[Compatible]	[OS = Service]	[Compatible]

Table: ..\..\data\oslan.asc Page: 2 Strip: 8

Transaction	Service	Subservice	Service »»

Table: ..\..\data\oslan.asc Page: 1 Strip: 9

Confined	Migrant	Singular	Plural
[Compatible]	[Remote Spawn]	[Compatible]	[Compatible]
[Compatible]	[Remote Fork]	[NA]	[Compatible]
[Compatible]	[Remote Prime]	[NA]	[Compatible]

Table: ..\..\data\oslan.asc Page: 2 Strip: 9

Confined	Migrant	Singular	Plural

Table: ..\..\data\oslan.asc Page: 1 Strip: 10

Distributed	Address »»	Broadcast	Circuit
	[Implements][Has]		
[Remote]	[Acquires]		
[Remote Spawn]	[Direct]	[Compatible]	[Compatible]
[Remote Fork]	[Via Parent]	[Via Parent]	[Via Parent]
[Remote Prime]	[Direct]	[Compatible]	[Compatible]

Table: ..\..\data\oslan.asc Page: 2 Strip: 10

Distributed	Address »»	Broadcast	Circuit

Table: ..\..\data\oslan.asc Page: 1 Strip: 11

Packet	Datagram	Message	Ticket
[Compatible]	[Compatible]	[Compatible]	[Incompatible]
[Via Parent]	[Via Parent]	[Incompatible]	[Incompatible]
[Compatible]	[Compatible]	[Compatible]	[Compatible]

Table: ..\..\data\oslan.asc Page: 2 Strip: 11

Packet	Datagram	Message	Ticket

Table: ..\..\data\oslan.asc Page: 1 Strip: 12

Transaction »»	Implicity	Transaction »»	Serialization »»
[Implements]		[Implements]	[Sequenced By]
[Initiates]	[Concurrent]	[Initiates]	[Sequential]
[Spawns][Enters]	[Concurrent Spawn]	[Spawns][Enters]	[Synchronous]
[Inherits][Nests]	[Concurrent Fork]	[Inherits][Nests]	[Synchronous]
[Begins][Enters]	[Compatible]	[Begins][Enters]	[Asynchronous]

Table: ..\..\data\oslan.asc Page: 2 Strip: 12

Transaction »»	Implicity	Transaction »»	Serialization »»

Table: ..\..\data\oslan.asc Page: 1 Strip: 13

Locks	Timestamps	Tickets	Transaction »»
			[Implements]
[Acquires]	[Acquires]	[Acquires]	[Initiates]
[Request Explicit]	[Given]	[Requests]	[Spawns][Enters]
[Inherit All]	[Inherits]	[Inherits]	[Inherits][Nests]
[Request Explicit]	[Given]	[Requests]	[Begins][Enters]

Table: ..\..\data\oslan.asc Page: 2 Strip: 13

Locks	Timestamps	Tickets	Transaction »»

Table: ..\..\data\oslan.asc Page: 1 Strip: 14

Logs »»	Undo	Intentions	Write-Ahead
[Writes][Examines]			
[Entered In]			
[Writes To]	[Unspawn]	[Spawn Request]	[2 Phase Spawn]
[Via Parent]	[Unfork]	[Fork Request]	[2 Phase Fork]
[Writes To]	[Compensate]	[Activate Request]	[2 Phase Activate]

Table: ..\..\data\oslan.asc Page: 2 Strip: 14

Logs »»	Undo	Intentions	Write-Ahead

Table: ..\..\data\oslan.asc Page: 1 Strip: 15

Transaction »»	Coordinate »»	Begin	Enter
[Implements]	[Obeys]	[Requests]	[Requests]
[Initiates]	[Request Initiate]	[Requests]	[Requests]
[Spawns][Enters]	[Request Spawn]	[Requests]	[Requests]
[Inherits][Nests]	[Request Fork]	[Incompatible]	[Nests]
[Begins][Enters]	[Request Activate]	[Requests]	[Requests]

Table: ..\..\data\oslan.asc Page: 2 Strip: 15

Transaction »»	Coordinate »»	Begin	Enter

Table: ..\..\data\oslan.asc Page: 1 Strip: 16

Nesting	Leave	Destiny	Transition
[Causes]	[Requests]	[Votes][Receives]	[Undergoes]
[Begins]	[Requests]	[Has]	[Incompatible]
[Begins]	[Requests]	[Has]	[Incompatible]
[Inherits]	[Unnests]	[Has]	[Incompatible]
[Begins]	[Requests]	[Has]	[Incompatible]

Table: ..\..\data\oslan.asc Page: 2 Strip: 16

Nesting	Leave	Destiny	Transition

Table: ..\..\data\oslan.asc Page: 1 Strip: 17

Commit/Abort	End	Transaction »	Restart »
[Implements]	[Implements]	[Implements]	[Implements]
[Begins]	[NA]	[Initiates]	[Initiates]
[Begins]	[NA]	[Spawns][Enters]	[Respawn]
[Begins]	[NA]	[Inherits][Nests]	[Refork]
[Begins]	[NA]	[Begins][Enters]	[Reactivate]

Table: ..\..\data\oslan.asc Page: 2 Strip: 17

Commit/Abort	End	Transaction »»	Restart »»

Table: ..\..\data\oslan.asc Page: 1 Strip: 18

ET Record	P1/P2 Record	Destiny ACK	Request ACK
[Examines]	[Examines]	[Examines]	[Examines]
[NA]	[NA]	[NA]	[NA]
[NA]	[NA]	[NA]	[NA]
[NA]	[NA]	[NA]	[NA]
[NA]	[NA]	[NA]	[NA]

Table: ..\..\data\oslan.asc Page: 2 Strip: 18

ET Record	P1/P2 Record	Destiny ACK	Request ACK

Table: ..\..\data\oslan.asc Page: 1 Strip: 19

— Request NAK —
[Examines]
[NA]
[NA]
[NA]
[NA]

Table: ..\..\data\oslan.asc Page: 2 Strip: 19

— Request NAK

A Knowledge Dictionary System for Scheduling Support

P.G. Ossorio and L.S. Schneider

Appendix D

Submitted by
Linguistic Research Institute, Inc.
5600 Arapahoe Avenue
Boulder, Colorado 80303

Submitted to
Rome Air Development Center
Griffiss AFB, New York

TABLE OF CONTENTS

1. Social Practice Description (SPD) Table.

Table: ..\..\data\schspd.asc Page: 1 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
1	0	#					<Acquire>		48	D	A	50		
1	0	#					[RP]		48	D	A	50		
1	0	#					[HW1]		48	D	A	50		
1	0	1					<PlanGet>	O						
1	0	1					[RP]	O						
1	0	1					[HW1]	O						
1	0	2			2		<Get>	M						
1	0	2			2		[RP]	M						
1	0	2			2		[HW1]	M						
1	0	2					<Verify>	O						
1	0	2					[RP]	O						
1	0	2					[HW1]	O						
2	0	#					<Acquire>							
2	0	#					[RP]							
2	0	#					[HW1Item]							
2	0	1					<Require>	O						
2	0	1					[RP]	O						
2	0	1					[HW1Item]	O						
2	0	2					<Contract for>	O						
2	0	2					[RP]	O						
2	0	2					[HW1Item]	O						
2	0	3					<Receive>	M						
2	0	3					[RP]	M						
2	0	3					[HW1Item]	M						
2	0	4					<Test>	O						
2	0	4					[RP]	O						
2	0	4					[HW1Item]	O						
2	0	5					<Accept>	M						
2	0	5					[RP]	M						
2	0	5					[HW1Item]	M						
2	0	6					<Cross off>	M						
2	0	6					[RP]	M						
2	0	6					[HW1Item]	M						
2	0	7			2		<Acquire>	R						
2	0	7			2		[RP]	R						
2	0	7			2		[HW1Item]	R						
3	1	#					<Create>		36	M	A	50		
3	1	#					[RP]							
3	1	#					[OS]							
3	2	#					<Acquire>		80	D	A	50		
3	2	#					[RP]		80	D	A	50		
3	2	#					[OS]		80	D	A	50		
3	2	1			31		<Create>	O						
3	2	1			31		[RP]	O						
3	2	1			31		[OSReq]	O						
3	2	2			32		<Develop>	O						
3	2	2			32		[RP]	O						
3	2	2			32		[OSList]	O						
3	2	3					<Select>	M						
3	2	3					[RP]	M						
3	2	3					[OS Vendor]	M						

Table: ..\..\data\schspd.asc Page: 2 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
3	2	4					<Purchase>	O						
3	2	4					[RP]	O						
3	2	4					[VenderOS]	O						
3	2	5					<Receive>	M						
3	2	5					[RP]	M						
3	2	5					[VendorOS]	M						
3	2	6					<Accept>	M						
3	2	6					[RP]	M						
3	2	6					[VendorOS]	M						
3	2	6					[OS]	M						
4	0	#					<Install>		60	D	A	50		
4	0	#					[RP]		60	D	A	50		
4	0	#					[HW2]		60	D	A	50		
4	0	#					[LocA]		60	D	A	50		
4	0	1					<Emplace>							
4	0	1					[RP]							
4	0	1					[HW2]							
4	0	1					[LocA]							
4	0	2					<Assemble>							
4	0	2					[RP]							
4	0	2					[HW2]							
4	0	2					[LocA]							
4	0	3					<Activate>							
4	0	3					[RP]							
4	0	3					[HW2]							
4	0	3					[LocA]							
4	0	4					<Test>							
4	0	4					[RP]							
4	0	4					[HW2]							
4	0	4					[LocA]							
4	0	5					<Accept>							
4	0	5					[RP]							
4	0	5					[HW2]							
4	0	5					[LocA]							
5	1	#					<Create>		36	M	A	50		
5	1	#					[RP]							
5	1	#					[NetSW]							
5	2	#					<Acquire>		80	D	A	50		
5	2	#					[RP]		80	D	A	50		
5	2	#					[NetSW]		80	D	A	50		
5	2	1			51		<Create>	O						
5	2	1			51		[RP]	O						
5	2	1			51		[NetSWReq]	O						
5	2	2			52		<Develop>	O						
5	2	2			52		[RP]	O						
5	2	2			52		[NetSWList]	O						
5	2	3					<Select>	M						
5	2	3					[RP]	M						
5	2	3					[NetSW Vendor]	M						
5	2	4					<Purchase>	O						
5	2	4					[RP]	O						

Table: ..\..\data\schspd.asc Page: 3 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
5	2	4					[VendorNetSW]	O						
5	2	5					<Receive>	M						
5	2	5					[RP]	M						
5	2	5					[VendorNetSW]	M						
5	2	6					<Accept>	M						
5	2	6					[RP]	M						
5	2	6					[VendorNetSW]	M						
5	2	6					[NetSW]	M						
6	0	#					<Acquire>		150	D	A	50		
6	0	#					[RP]		150	D	A	50		
6	0	#					[AppSW]		150	D	A	50		
6	0	1					<PlanGet>	O						
6	0	1					[RP]	O						
6	0	1					[AppSW]	O						
6	0	2			7		<Get>	M						
6	0	2			7		[RP]	M						
6	0	2			7		[AppSW]	M						
6	0	3					<Verify>	O						
6	0	3					[RP]	O						
6	0	3					[AppSW]	O						
7	0	#					<Acquire>							
7	0	#					[RP]							
7	0	#					[AppSWItem]							
7	0	1					<Require>	O						
7	0	1					[RP]	O						
7	0	1					[AppSWItem]	O						
7	0	2					<Contract for>	O						
7	0	2					[RP]	O						
7	0	2					[AppSWItem]	O						
7	0	3					<Receive>	M						
7	0	3					[RP]	M						
7	0	3					[AppSWItem]	M						
7	0	4					<Test>	O						
7	0	4					[RP]	O						
7	0	4					[AppSWItem]	O						
7	0	5					<Accept>	M						
7	0	5					[RP]	M						
7	0	5					[AppSWItem]	M						
7	0	6					<Cross off>	M						
7	0	6					[RP]	M						
7	0	6					[AppSWItem]	M						
7	0	7			7		<Acquire>	R						
7	0	7			7		[RP]	R						
7	0	7			7		[AppSWItem]	R						
8	0	#					<Install>		90	D	A	50		
8	0	#					[RP]		90	D	A	50		
8	0	#					[OS]		90	D	A	50		
8	0	#					[HW1]		90	D	A	50		
8	0	1					<DoLoad>							
8	0	1					[RP]							
8	0	1					[OS]							

Table: ..\..\data\schspd.asc Page: 4 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
8	0	1					[HW1]							
8	0	2			9		<ValTest>							
8	0	2			9		[RP]							
8	0	2			9		[OS]							
8	0	2			9		[HW1]							
8	0	3					<Evaluate>							
8	0	3					[RP]							
8	0	3					[ValTest]							
8	0	4					<Accept>							
8	0	4					[RP]							
8	0	4					[OS]							
9	0	#					<ValTest>							
9	0	#					[RP]							
9	0	#					[OS]							
9	0	#					[HW1]							
9	0	#					[ListCItem] * UNDEF *							
9	0	1					<Select>							
9	0	1					[RP]							
9	0	1					[ListCItem] * UNDEF *							
9	0	2					<Load>							
9	0	2					[RP]							
9	0	2					[ListCItem] * UNDEF *							
9	0	2					[OS]							
9	0	2					[HW1]							
9	0	3					<Operate>							
9	0	3					[RP]							
9	0	3					[ListCItem] * UNDEF *							
9	0	3					[OS]							
9	0	3					[HW1]							
9	0	4					<Examine>							
9	0	4					[RP]							
9	0	4					[ListCItem] * UNDEF *							
9	0	4					[OS]							
9	0	4					[HW1]							
9	0	5					<Cross off>							
9	0	5					[RP]							
9	0	5					[ListCItem] * UNDEF *							
9	0	6					<ValTest>	R						
9	0	6					[RP]	R						
9	0	6					[ListCItem] * UNDEF *	R						
9	0	6					[OS]	R						
9	0	6					[HW1]	R						
10	0	#					<Install>							
10	0	#					[RP]							
10	0	#					[NetSW]							
10	0	#					[HW1]							
10	0	1					<DoLoad>							
10	0	1					[RP]							
10	0	1					[NetSW]							
10	0	1					[HW1]							
10	0	2			9		<ValTest>							

Table: ..\..\data\schspd.asc Page: 5 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
10	0	2			9		[RP]							
10	0	2			9		[NetSW]							
10	0	2			9		[HW1]							
10	0	3					<Evaluate>							
10	0	3					[RP]							
10	0	3					[ValTest]							
10	0	4					<Accept>							
10	0	4					[RP]							
10	0	4					[NetSW]							
11	0	#					<ValTest>							
11	0	#					[RP]							
11	0	#					[NetSW]							
11	0	#					[HW1]							
11	0	#					[ListDItem] * UNDEF *							
11	0	1					<Select>							
11	0	1					[RP]							
11	0	1					[ListDItem] * UNDEF *							
11	0	2					<Load>							
11	0	2					[RP]							
11	0	2					[ListDItem] * UNDEF *							
11	0	2					[NetSW]							
11	0	2					[HW1]							
11	0	3					<Operate>							
11	0	3					[RP]							
11	0	3					[ListDItem] * UNDEF *							
11	0	3					[NetSW]							
11	0	3					[HW1]							
11	0	4					<Examine>							
11	0	4					[RP]							
11	0	4					[ListDItem] * UNDEF *							
11	0	4					[NetSW]							
11	0	4					[HW1]							
11	0	5					<Cross off>							
11	0	5					[RP]							
11	0	5					[ListDItem] * UNDEF *							
11	0	6					<ValTest>	R						
11	0	6					[RP]	R						
11	0	6					[ListDItem] * UNDEF *	R						
11	0	6					[NetSW]	R						
11	0	6					[HW1]	R						
31	0	#					<Add to>							
31	0	#					[RP]							
31	0	#					[OSReqItem]							
31	0	1					<Select>							
31	0	1					[RP]							
31	0	1					[OSNeed]							
31	0	1					[ListN]							
31	0	2					<Analyze>							
31	0	2					[RP]							
31	0	2					<OSNeed>							
31	0	3					<Cross off>							

Table: ..\..\data\schspd.asc Page: 6 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
31	0	3					[RP]							
31	0	3					[OSNeed]							
31	0	3					[ListN]							
31	0	4			31		<Add to>	R						
31	0	4			31		[RP]	R						
31	0	4			31		[OSReqItem]	R						
32	0	#					<Add to>							
32	0	#					[RP]							
32	0	#					[OSListItem]							
32	0	1					<Select>							
32	0	1					[RP]							
32	0	1					[VendorOS]							
32	0	1					[ListP]							
32	0	2			33		<Compare>							
32	0	2			33		[RP]							
32	0	2			33		[OSReq]							
32	0	2			33		[VendorOS]							
32	0	3					<Cross off>							
32	0	3					[RP]							
32	0	3					[VendorOS]							
32	0	3					[ListP]							
32	0	4					<Add to>							
32	0	4					[RP]	R						
32	0	4					[VendorOS]	R						
32	0	4					[OSListItem]	R						
33	0	#					<Compare>							
33	0	#					[RP]							
33	0	#					[OSReqItem]							
33	0	#					[VendorOS]							
33	0	1					<Select>							
33	0	1					[RP]							
33	0	1					[OSReqItem]							
33	0	2					<Cross off>							
33	0	2					[RP]							
33	0	2					[OSReqItem]							
33	0	3			33		<Compare>	R						
33	0	3			33		[RP]	R						
33	0	3			33		[OSReqItem]	R						
33	0	3			33		[VendorOS]	R						
50	0	#					<Create>		0	D	A	LC-I		
50	0	#					[RP]		0	D	A	LC-I		
50	0	#					[DHS]		0	D	A	LC-I		
50	0	1			1		<Acquire>		48	D	A	LC-I		
50	0	1			1		[RP]		48	D	A	LC-I		
50	0	1			1		[HW1]		48	D	A	LC-I		
50	0	2			4		<Install>		60	D	A	LC-I		
50	0	2			4		[RP]		60	D	A	LC-I		
50	0	2			4		[HW1]		60	D	A	LC-I		
50	0	2			4		[LocA]		60	D	A	LC-I		
50	0	3	A		8		<Install>		90	D	A	LC-I		
50	0	3	A		8		[RP]		90	D	A	LC-I		

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
50	0	3	A		8		[HW2]		90	D	A	LC-I		
50	0	3	A		8		[OS]		90	D	A	LC-I		
50	0	3	B		10		<Install>		150	D	A	LC-I		
50	0	3	B		10		[RP]		150	D	A	LC-I		
50	0	3	B		10		[HW2]		150	D	A	LC-I		
50	0	3	B		10		[NetSW]		150	D	A	LC-I		
50	0	4					<Install>		210	D	A	LC-I		
50	0	4					[RP]		210	D	A	LC-I		
50	0	4					[HW2]		210	D	A	LC-I		
50	0	4					[AppSW]		210	D	A	LC-I		
50	0	5					<Install>		270	D	A	LC-I		
50	0	5					[RP]		270	D	A	LC-I		
50	0	5					[DHS]		270	D	A	LC-I		
50	0	5					[ListK] * UNDEF *		270	D	A	LC-I		
50	0	6	A				<Certifies>		300	D	A	LC-I		
50	0	6	A				[RPC]		300	D	A	LC-I		
50	0	6	A				[DHS]		300	D	A	LC-I		
50	0	6	B				<Accepts>		305	D	A	LC-I		
50	0	6	B				[UserRP]		305	D	A	LC-I		
50	0	6	B				[DHS]		305	D	A	LC-I		
50	0	¦	A		3		<Acquire>		80	D	A	LC-I		
50	0	¦	A		3		[RP]		80	D	A	LC-I		
50	0	¦	A		3		[OS]		80	D	A	LC-I		
50	0	¦	B		5		<Acquire>		110	D	A	LC-I		
50	0	¦	B		5		[RP]		110	D	A	LC-I		
50	0	¦	B		5		[NetSW]		110	D	A	LC-I		
50	0	¦	C		6		<Acquire>		150	D	A	LC-I		
50	0	¦	C		6		[RP]		150	D	A	LC-I		
50	0	¦	C		6		[AppSW]		150	D	A	LC-I		
51	0	#					<Add to>							
51	0	#					[RP]							
51	0	#					[NewSWReqItem]							
51	0	1					<Select>							
51	0	1					[RP]							
51	0	1					[NewSWNeed]							
51	0	1					[ListN]							
51	0	2					<Analyze>							
51	0	2					[RP]							
51	0	2					<NewSWNeed>							
51	0	3					<Cross off>							
51	0	3					[RP]							
51	0	3					[NewSWNeed]							
51	0	3					[ListN]							
51	0	4			51		<Add to>	R						
51	0	4			51		[RP]	R						
51	0	4			51		[NewSWReqItem]	R						
52	0	#					<Add to>							
52	0	#					[RP]							
52	0	#					[NewSWListItem]							
52	0	1					<Select>							
52	0	1					[RP]							

Table: ..\..\data\schspd.asc Page: 8 Strip: 1

PRO	PAR	STG	SS	OPT	OS	FTYPE	ELEMENT	MO	DTG	DTU	TR	PROD	TS	\
52	0	1					[VendorNewSW]							
52	0	1					[ListP]							
52	0	2			53		<Compare>							
52	0	2			53		[RP]							
52	0	2			53		[NewSWReq]							
52	0	2			53		[VendorNewSW]							
52	0	3					<Cross off>							
52	0	3					[RP]							
52	0	3					[VendorNewSW]							
52	0	3					[ListP]							
52	0	4					<Add to>							
52	0	4					[RP]	R						
52	0	4					[VendorNewSW]	R						
52	0	4					[NewSWListItem]	R						
53	0	#					<Compare>							
53	0	#					[RP]							
53	0	#					[NewSWReqItem]							
53	0	#					[VendorNewSW]							
53	0	1					<Select>							
53	0	1					[RP]							
53	0	1					[NewSWReqItem]							
53	0	2					<Cross off>							
53	0	2					[RP]							
53	0	2					[NewSWReqItem]							
53	0	3			53		<Compare>	R						
53	0	3			53		[RP]	R						
53	0	3			53		[NewSWReqItem]	R						
53	0	3			53		[VendorNewSW]	R						

2. Element Individual List (EIL) Table.

Table: ..\..\data\scheil.asc Page: 1 Strip: 1

FRQ	PAR	STG	SS	OPT	OS	ELEMENT	INDIVIDUAL
1	0	#				[RPParm1]	[RPIBM]
1	0	#				[RPParm1]	IBM
1	0	#				[HW1]	[ListA]
1	0	#		1	A	[ListA]	[Processor]
1	0	#		1	A	[ListA]	[Console]
1	0	#		1	A	[ListA]	[Network Hardware]
1	0	#		1	A	[ListA]	[Extended Memory Interface]
1	0	#		1	A	[ListA]	[Peripheral Interface]
1	0	#		1	A	[ListA]	[Mass Storage]
1	0	#		1	A	[ListA]	[Peripherals]
1	0	#		1	B	[Processor]	[CPU]
1	0	#		1	B	[Processor]	[FPU]
1	0	#		1	B	[Processor]	[Memory Bus]
1	0	#		1	C	[Console]	[Monitor]
1	0	#		1	C	[Console]	[Video Interface]
1	0	#		1	C	[Console]	[Keyboard]
1	0	#		1	C	[Console]	[Mouse]
1	0	#		1	D	[Network Hardware]	[Network Backplane]
1	0	#		1	D	[Network Hardware]	[Network Interface Board]
1	0	#		1	D	[Network Hardware]	[Gateway Board]
1	0	#		1	E	[Mass Storage]	[Mass Storage Interface]
1	0	#		1	E	[Mass Storage]	[Mass Storage Devices]
1	0	#		1	F	[Peripherals]	[Printer]
1	0	#		1	F	[Peripherals]	[Tape Drive]
2	0	#				[RPParm1]	[RPIBM]
2	0	#				[RPParm1]	IBM
2	0	#				[HW1]	[ListA]
2	0	#		1	A	[ListA]	[Processor]
2	0	#		1	A	[ListA]	[Console]
2	0	#		1	A	[.istA]	[Network Hardware]
2	0	#		1	A	[ListA]	[Extended Memory Interface]
2	0	#		1	A	[ListA]	[Peripheral Interface]
2	0	#		1	A	[ListA]	[Mass Storage]
2	0	#		1	A	[ListA]	[Peripherals]
2	0	#				[CPU]	National 32032
2	0	#				[CPU]	Motorola 68030
2	0	#				[CPU]	Intel 80387
2	0	#				[FPU]	Motorola 68881
2	0	#				[FPU]	Motorola 68882
2	0	#				[FPU]	Intel 80387
2	0	#				[FPU]	Intel 80287
2	0	#				[Memory Bus]	Unibus
2	0	#				[Memory Bus]	Versabus
2	0	#				[Memory Bus]	MicroChannel
2	0	#				[Peripheral Interface]	[RS-232]
2	0	#				[Peripheral Interface]	[IEEE-488]
2	0	#				[Peripheral Interface]	[Centronics]
2	0	#				[Monitor]	[Monochrome Monitor]
2	0	#				[Monitor]	[Color Composite Monitor]
2	0	#				[Monitor]	[RGB Monitor]
2	0	#				[Monitor]	[XY Monitor]

Table: ..\..\data\scheil.asc Page: 2 Strip: 1

PRO	PAR	STG	SS	OPT	OS	ELEMENT	INDIVIDUAL
2	0	#				[Monitor]	[HiRes Monitor]
2	0	#				[Video Interface]	[MDA Interface]
2	0	#				[Video Interface]	[CGA Interface]
2	0	#				[Video Interface]	[EGA Interface]
2	0	#				[Video Interface]	[VGA Interface]
2	0	#				[Keyboard]	[XT Type Keyboard]
2	0	#				[Keyboard]	[AT Type Keyboard]
2	0	#				[Keyboard]	[Mouse]
2	0	#				[Mass Storage Interface]	[SCSI]
2	0	#				[Mass Storage Interface]	[WDI]
2	0	#				[Mass Storage Devices]	[Fixed Hard Disk]
2	0	#				[Mass Storage Devices]	[Disk Cartridge]
2	0	#				[Mass Storage Devices]	[CD-ROM]
2	0	#				[Mass Storage Devices]	[CD-WORM]
2	0	#				[Printer]	[NLQ Printer]
2	0	#				[Printer]	[High Speed Printer]
2	0	#				[Printer]	[Laser Printer]
2	0	#				[Tape Drive]	[Start/Stop Tape Drive]
2	0	#				[Tape Drive]	[Streamer Tape Drive]
2	0	#				[Network Interface Board]	G-Net Board
2	0	#				[Network Interface Board]	Omninet Board
2	0	#				[Network Interface Board]	Arcnet Board
2	0	#				[Network Interface Board]	3COM Board
2	0	#				[Network Interface Board]	Plan 2000 Board
2	0	#				[Network Backplane]	[Optical Bus Backplane]
2	0	#				[Network Backplane]	[Optical Ring Backplane]
2	0	#				[Network Backplane]	[Optical Star Backplane]
2	0	#				[Network Backplane]	[Optical Cluster Backplane]
2	0	#				[Network Backplane]	[Coaxial Bus Backplane]
2	0	#				[Network Backplane]	[Coaxial Ring Backplane]
2	0	#				[Network Backplane]	[Coaxial Star Backplane]
2	0	#				[Network Backplane]	[Coaxial Cluster Backplane]
2	0	#				[Network Backplane]	[Twisted Pair Bus Backplane]
2	0	#				[Network Backplane]	[Twisted Pair Ring Backplane]
2	0	#				[Network Backplane]	[Twisted Pair Star Backplane]
2	0	#				[Network Backplane]	[Twisted Pair Cluster Backplane]
2	0	#				[Network Backplane]	[VHF Bus Backplane]
2	0	#				[Network Backplane]	[UHF Bus Backplane]
2	0	#				[Gateway Board]	[X.25 Gateway Board]
2	0	#				[Gateway Board]	[SNA Gateway Board]
2	0	#				[Gateway Board]	[SDLC Gateway Board]
3	0	#				[OS]	Unix System V
3	0	#				[OS]	Xenix 387
3	0	#				[OS]	QNX
3	0	#				[OS]	IRMX 87
3	0	#				[OS]	[UCSD p]
3	0	#				[OS]	Concurrent
3	0	#				[OS]	VM/PC
3	0	#				[OS]	CX/PC
3	0	#				[OS]	MSDOS 5.x
3	0	#				[NetSW]	Netware

Table: ..\..\data\scheil.asc Page: 3 Strip: 1

PRO	PAR	STG	SS	OPT	OS	ELEMENT	INDIVIDUAL
3	0	#				[NetSW]	Grapevine
3	0	#				[NetSW]	Locus
3	0	#				[NetSW]	3 Plus
3	0	#				[NetSW]	Omninet
3	0	#				[NetSW]	PC-Net
3	0	#				[NetSW]	Multilink
3	0	#				[NetSW]	Tiara Link
3	0	#				[NetSW]	Tapestry
3	0	#				[NetSW]	ARCnet
4	0	#				[HW2]	[ListR]
4	0	#		1	R	[ListR]	[Processor]
4	0	#		1	R	[ListR]	[Console]
4	0	#		1	R	[ListR]	[Network Hardware]
4	0	#		1	R	[ListR]	[Extended Memory Interface]
4	0	#		1	R	[ListR]	[Peripheral Interface]
4	0	#		1	R	[ListR]	[Mass Storage]
4	0	#		1	R	[ListR]	[Peripherals]
5	0	#		1	NS	[NetSWReqs]	[Bandwidth Req]
5	0	#		1	NS	[NetSWReqs]	[MT Failures Req]
5	0	#		1	NS	[NetSWReqs]	[MT Recovery Req]
5	0	#		1	NS	[NetSWReqs]	[Failure Mode Req]
5	0	#		1	NS	[NetSWReqs]	[Compatibilities Req]
5	0	#		1	NS	[NetSWReqs]	[Services Req]
5	0	#		1		[Bandwidth Req]	etc
6	0	#				[AppSW]	[ListB]
6	0	#		1	B	[ListB]	[Text Editor]
6	0	#		1	B	[ListB]	[Word Processor]
6	0	#		1	B	[ListB]	[File Manage SW]
6	0	#		1	B	[ListB]	[Spreadsheet SW]
6	0	#		1	B	[ListB]	[Integrated SW]
6	0	#		1	B	[ListB]	[Compilers]
6	0	#		1		[Compilers]	[Pascal Compiler]
6	0	#		1		[Compilers]	[Ada Compiler]
6	0	#		1		[Compilers]	[C Compiler]
6	0	#		1		[Compilers]	[Prolog Compiler]
6	0	#		1		[Compilers]	[Basic Compiler]
7	0	#				[Text Editor]	Vedit
7	0	#				[Text Editor]	[Vi]
7	0	#				[Text Editor]	TT
7	0	#				[Text Editor]	[SPF]
7	0	#				[Word Process SW]	Wordstar
7	0	#				[Word Process SW]	Word Perfect
7	0	#				[Word Process SW]	Word
7	0	#				[File Manage SW]	ProFile
7	0	#				[File Manage SW]	Hyper Card
7	0	#				[File Manage SW]	TT
7	0	#				[Database Manage SW]	Paradox
7	0	#				[Database Manage SW]	MDBS
7	0	#				[Database Manage SW]	Pillar
7	0	#				[Database Manage SW]	TT
7	0	#				[Spreadsheet SW]	SuperCalc

Table: ..\..\data\scheil.asc Page: 4 Strip: 1

PRO	PAR	STG	SS	OPT	OS	ELEMENT	INDIVIDUAL
7	0	#				[Spreadsheet SW]	MultiPlan
7	0	#				[Spreadsheet SW]	TK!Solver
7	0	#				[Spreadsheet SW]	Lotus 1-2-3
7	0	#				[Integrated SW]	Symphony
7	0	#				[Integrated SW]	Framework II
7	0	#				[Pascal Compiler]	Borland Pascal Compiler
7	0	#				[Pascal Compiler]	Microsoft Pascal Compiler
7	0	#				[Pascal Compiler]	[UCSD Pascal Compilerr
7	0	#				[Ada Compiler]	Alsys Ada Compiler
7	0	#				[Ada Compiler]	Janus Ada Compiler
7	0	#				[C Compiler]	Borland C Compiler
7	0	#				[C Compiler]	Lattice C Compiler
7	0	#				[C Compiler]	Microsoft C Compiler
7	0	#				[Prolog Compiler]	Arity Prolog Compiler
7	0	#				[Prolog Compiler]	Borland Prolog Compiler
7	0	#				[Basic Compiler]	Borland Basic Compiler
7	0	#				[Basic Compiler]	True Basic Compiler

A Knowledge Dictionary System for Scheduling Support

P.G. Ossorio and L.S. Schneider

Appendix E

Submitted by
Linguistic Research Institute, Inc.
5600 Arapahoe Avenue
Boulder, Colorado 80303

Submitted to
Rome Air Development Center
Griffiss AFB, New York

TABLE OF CONTENTS

1. Fact Type Table

FTYPE	DESCRIPTION
1	[Process] <Started> [On Time]
2	[Process] <Started> [Late]
3	[Process] <Started> [Not Yet]
4	[Process] <Proceeding> [Per Schedule]
5	[Process] <Proceeding> [Behind Schedule]
6	[Process] <Estimate> [Completion]
7	[Process] <Estimate> [Start]
8	[Process] <Completed>
9	[Process] <Started> [Just Now]
10	[Process] <Completed> [Just Now]
11	[Process] <Started> [Early]
12	[Process] <Completed> [Late]
13	[Process] <Completed> [Early]
14	[Process] <Delayed By> [Process]
15	[Process] <Delayed By> [Object]
21	[RP] <Create> [Requirements] [OS]
22	[RP] <Create> [Requirements] [NetSW]
23	[RP] <Create> [Requirements] [AppSW]
24	[RP] <Create> [DHS Node] [Loc]
31	[RP] <Assemble> [HW2] [Loc]
41	[RP] <Contract> [Purchase] [OS] [VenderOS]
51	[RP] <Receive> [OS] [VenderOS]
61	[RPM] <Accept> [OS]
71	[RP] <Select> [NetSW]
81	[RP] <DoLoad> [NetSW] [OS] [HW2]
91	[RP] <ValTest> [NetSW] [OS] [HW2]
101	[RP] <Verify> [AppSW]
111	[RPM] <Certify> [DHS]
112	[DHS] <Uncertified By> [Constituent]
121	[RPUser] <Accept> [DHS]

2. Fact Table Page: 1 Strip: 1

ID	FTYPE	ELEMENT	DATE
1	1	[50] <Started> [On Time]	890301
2	4	[01] <Proceeding> [Per Schedule]	890401
4	4	[01.0.2] <Proceeding> [Per Schedule]	890328
5	8	[01] <Completed>	890415
6	21	[RPTRW] <Create> [Requirements] [OS]	890401
7	31	[IBM] <Assemble> [DHS Hardware] [Omaha]	890425
8	4	[04] <Proceeding> [Per Schedule]	
9	10	[04] <Completed> [Just Now]	890430
10	41	[TRW] <Contract> [Purchase] [Unix][ISC]	890415
11	51	[TRW] <Receive> [Unix] [ISC]	890510
12	51	[TRW] <Receive> [Unix] [Bell]	890510
13	22	[DEC] <Create> [Requirements] [NetSW]	
14	61	[RPOmaha] <Accept> [Unix]	890506
15	71	[RPDEC] <Select> [DECNET]	890605
16	81	[DEC] <DoLoad> [DECNET] [Unix] [DHS]	890701
17	91	[DEC] <ValTest> [DECNET] [Unix] [DHS]	890720
18	4	[10] <Proceeding> [Per Schedule]	890720
19	6	[10] <Estimate> [Completion]	890901
20	6	[51..5] <Estimate> [Completion]	890915
21	6	[10] <Estimate> [Completion]	890915
22	6	[06] <Estimate> [Completion]	890901
23	101	[P. J. Aucoin] <Verify> [AppSW]	890901
24	6	[51..5] <Estimate> [Completion]	891215
25	6	[51..5] <Estimate> [Completion]	891201
26	10	[51..5] <Completed> [Just Now]	891101
27	14	[51] <Delayed By> [User Interface]	891101
28	10	[51] <Completed> [Just Now]	
29	6	[50..2] <Estimate> [Completion]	900101
30	14	[50..2] <Delayed By> []	900101
31	8	[50..2] <Completed>	900115
32	112	[DHS] <Uncertified By> [User Interface]	900115
33	111	[RPM] <Certify> [DHS User Interface]	900130
34	121	[RPUser] <Accept> [DHS]	900130

Page: 1 Strip: 2

SOURCE	RDATE	PR	EXPLANATION
RPIBM	890310	3	Contract F30602-C-89-117
RPIBM	890404	3	
RPIBM	890406	3	
H. R. Robinson	890420	3	
M. J. O'Brien	890420	2	
H. R. Robinson	890425	3	
K. H. Martinez	890425	3	
H. R. Robinson	890430	3	
M. J. O'Brien	890430	2	
M. J. O'Brien	890430	2	Per Purchase Contract
M. J. O'Brien	890515	2	
K. C. Jones	890601	3	
M. J. O'Brien	890610	2	OS Installation Complete
K. C. Jones	890615	3	
K. C. Jones	890705	3	
K. C. Jones	890725	3	
K. C. Jones	890725	3	
K. C. Jones	890801	3	Compatibility Problem
M. J. O'Brien	890805	2	Compatibility Problem
K. C. Jones	890815	3	Compatibility Problem
M. J. O'Brien	890815	2	
M. J. O'Brien	890901	2	
M. J. O'Brien	890901	2	Delay in Acquiring SW
M. J. O'Brien	890915	2	Revised Completion Estimate
M. J. O'Brien	891105	2	
M. J. O'Brien	891105	2	
M. J. O'Brien	891110	2	
M. J. O'Brien	891201	2	Delay in First Installation
M. J. O'Brien	900101		Delay in First Installation
M. J. O'Brien	900115		Delay in First Installation
TRW	900201	2	
TRW	900210	2	User Interface Certified
TRW	900210	2	

Page: 1 Strip: 3

RPPSO	RFACT	RSCHED
50..3		30 day delay
50..3	19	45 day delay
50..3	19	45 day delay
		30 day delay
06..3		
51	23	45 day delay
	24	30 day delay
	25	30 day delay
		30 day delay plus x
		40 day delay
	28	30 day delay
		30 day delay plus x
	28	45 day delay
		15 day delay plus x
	32	30 day delay
		30 day delay

MISSION

of

Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.